AIX Version 7.2

Base Operating System (BOS) Runtime Services

Note

Before using this information and the product it supports, read the information in “Notices” on page 2387.

This edition applies to AIX Version 7.2 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this document

This document provides users and system administrators with complete information about AIX Base Operating System Runtime Services.

Highlighting

The following highlighting conventions are used in this document:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-sensitivity in AIX

Everything in the AIX® operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type LS, the system responds that the command `is not found`. Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

Base Operating System (BOS) Runtime Services

This topic collection contains information, for experienced C programmers, about system calls, subroutines, functions, macros, and statements associated with base operating system runtime services.

The AIX operating system is designed to support The Open Group's Single UNIX Specification Version 3 (UNIX 03) for portability of operating systems based on the UNIX operating system. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. To determine the correct way to develop a UNIX 03 portable application, see The Open Group's UNIX 03 specification on The UNIX System website (http://www.unix.org).

What's new in AIX 7.2

Read about new or significantly changed information for the AIX 7.2 operating system.

How to see what's new or changed

To help you see where technical changes have been made, the AIX 7.2 information uses:

- The >| image to mark where new or changed information begins.
- The |< image to mark where new or changed information ends.

November 2020

The following information is a summary of the updates made to this topic collection:

- Added information about the `locobj` argument in the duplocale topic.
- Added information about the `EOVERFLOW` error code in the printf topic.
- Update the information about the `RLIMIT_AS` resource parameter in the getrlimit_64 and kgettrlimit64 topics.
- Added information about the pthread_mutexattr_getrobust_setrobust and pthread_mutex_consistent. Updated information about the behavior description for the robust mutex and added new error code values in the pthread_mutex_lock and pthread_mutex_timedlock topics. Also, added new error code values in the pthread_mutex_getprioceiling and pthread_cond_wait topics.
- Added information about the `pollest.h` header file in the pollset topic. The file defines structures and flags that are used by `pollset` subroutines.
- Updated information about the items %c, %y and %Y in the strptime topic.

November 2019

The following information is a summary of the updates made to this topic collection:

- Updated information about the `ffsl()`, `ffsll()` APIs and `libc` in the bcopy, bcmp, bzero, ffs, ffsl, or ffsll Subroutine topic.
- Added information about a new subroutine in pthread_rwlock_attr_setfavorwriters_np or pthread_rwlock_attr_getfavorwriters_np Subroutine topic.
- Added information about the **`strftime_l`** subroutine in the strftime or strftime_l Subroutine topic.

a

The following Base Operating System (BOS) runtime services begin with the letter *a*.

a64l or l64a Subroutine

Purpose

Converts between long integers and base-64 ASCII strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

long a64l (*String*)
char ***String**;

char *l64a (*LongInteger*)
long *LongInteger*;

Description

The **a64l** and **l64a** subroutines maintain numbers stored in base-64 ASCII characters. This is a notation in which long integers are represented by up to 6 characters, each character representing a digit in a base-64 notation.

The following characters are used to represent digits:

Character	Description
.	Represents 0.
/	Represents 1.
0 -9	Represents the numbers 2-11.
A-Z	Represents the numbers 12-37.
a-z	Represents the numbers 38-63.

Parameters

Item	Description
String	Specifies the address of a null-terminated character string.
LongInteger	Specifies a long value to convert.

Return Values

The **a64l** subroutine takes a pointer to a null-terminated character string containing a value in base-64 representation and returns the corresponding **long** value. If the string pointed to by the *String* parameter contains more than 6 characters, the **a64l** subroutine uses only the first 6.

Conversely, the **l64a** subroutine takes a **long** parameter and returns a pointer to the corresponding base-64 representation. If the *LongInteger* parameter is a value of 0, the **l64a** subroutine returns a pointer to a null string.

The value returned by the **l64a** subroutine is a pointer into a static buffer, the contents of which are overwritten by each call.

If the **String* parameter is a null string, the **a64l** subroutine returns a value of 0L.

If *LongInteger* is 0L, the **l64a** subroutine returns a pointer to a null string.

abort Subroutine

Purpose

Sends a **SIGIOT** signal to end the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
int abort (void)
```

Description

The **abort** subroutine sends a **SIGIOT** signal to the current process to terminate the process and produce a memory dump. If the signal is caught and the signal handler does not return, the **abort** subroutine does not produce a memory dump.

If the **SIGIOT** signal is neither caught nor ignored, and if the current directory is writable, the system produces a memory dump in the **core** file in the current directory and prints an error message.

The abnormal-termination processing includes the effect of the **fclose** subroutine on all open streams and message-catalog descriptors, and the default actions defined as the **SIGIOT** signal. The **SIGIOT** signal is sent in the same manner as that sent by the **raise** subroutine with the argument **SIGIOT**.

The status made available to the **wait** or **waitpid** subroutine by the **abort** subroutine is the same as a process terminated by the **SIGIOT** signal. The **abort** subroutine overrides blocking or ignoring the **SIGIOT** signal.

Note: The **SIGABRT** signal is the same as the **SIGIOT** signal.

Return Values

The **abort** subroutine does not return a value.

abs, div, labs, ldiv, imul_dbl, umul_dbl, llabs, or lldiv Subroutine

Purpose

Computes absolute value, division, and double precision multiplication of integers.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
int abs ( i  )
int i;
```

```
#include <stdlib.h>
```

```
long labs ( i )
long i;
```

```
#include <stdlib.h>
```

```
div_t div ( Numerator,  Denominator)
int Numerator: Denominator;
```

```
#include <stdlib.h>
```

```
void imul_dbl ( i,  j,  Result)
long i, j;
long *Result;
```

```
#include <stdlib.h>
```

```
ldiv_t ldiv (Numerator, Denominator)
long Numerator: Denominator;
```

```
#include <stdlib.h>
```

```
void umul_dbl (i, j, Result)
unsigned long i, j;
unsigned long *Result;
```

```
#include <stdlib.h>
```

```
long long int llabs(i)
long long int i;
```

```
#include <stdlib.h>
```

```
lldiv_t lldiv (Numerator, Denominator)
long long int Numerator, Denominator;
```

Description

The **abs** subroutine returns the absolute value of its integer operand.

Note: A twos-complement integer can hold a negative number whose absolute value is too large for the integer to hold. When given this largest negative value, the **abs** subroutine returns the same value.

The **div** subroutine computes the quotient and remainder of the division of the number represented by the *Numerator* parameter by that specified by the *Denominator* parameter. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented (for example, if the denominator is 0), the behavior is undefined.

The **labs** and **ldiv** subroutines are included for compatibility with the ANSI C library, and accept long integers as parameters, rather than as integers.

The **imul_dbl** subroutine computes the product of two signed longs, *i* and *j,* and stores the double long product into an array of two signed longs pointed to by the *Result* parameter.

The **umul_dbl** subroutine computes the product of two unsigned longs, *i* and *j,* and stores the double unsigned long product into an array of two unsigned longs pointed to by the *Result* parameter.

The **llabs** and **lldiv** subroutines compute the absolute value and division of long long integers. These subroutines operate under the same restrictions as the **abs** and **div** subroutines.

Note: When given the largest negative value, the **llabs** subroutine (like the **abs** subroutine) returns the same value.

Parameters

Item	Description
i	Specifies, for the **abs** subroutine, some integer; for **labs** and **imul_dbl,** some long integer; for the **umul_dbl** subroutine, some unsigned long integer; for the **llabs** subroutine, some long long integer.
Numerator	Specifies, for the **div** subroutine, some integer; for the **ldiv** subroutine, some long integer; for **lldiv**, some long long integer.
j	Specifies, for the **imul_dbl** subroutine, some long integer; for the **umul_dbl** subroutine, some unsigned long integer.
Denominator	Specifies, for the **div** subroutine, some integer; for the **ldiv** subroutine, some long integer; for **lldiv**, some long long integer.
Result	Specifies, for the **imul_dbl** subroutine, some long integer; for the **umul_dbl** subroutine, some unsigned long integer.

Return Values

The **abs**, **labs**, and **llabs** subroutines return the absolute value. The **imul_dbl** and **umul_dbl** subroutines have no return values. The **div** subroutine returns a structure of type **div_t**. The **ldiv** subroutine returns a structure of type **ldiv_t**, comprising the quotient and the remainder. The structure is displayed as:

```
struct ldiv_t {
  int quot; /* quotient */
  int rem;  /* remainder */
};
```

The **lldiv** subroutine returns a structure of type **lldiv_t**, comprising the quotient and the remainder.

access, accessx, faccessx, accessxat, or faccessat Subroutine

Purpose

Determines the accessibility of a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

int access (*PathName*,*Mode*)
char **PathName*;

```
int Mode;

int accessx (PathName,  Mode,  Who)
char *PathName;
int Mode, Who;

int faccessx (FileDescriptor, Mode, Who)
int FileDescriptor;
int Mode, Who;

int accessxat (DirFileDescriptor, PathName, Mode, Who)
int DirFileDescriptor;
char *PathName;
int Mode, Who;

int faccessat (DirFileDescriptor, PathName, Mode, Flag)
int DirFileDescriptor;
char *PathName;
int Mode, Flag;
```

Description

The **access**, **accessx**, **accessxat**, **faccessat** and **faccessx** subroutines determine the accessibility of a file system object. The **accessx**, **accessxat**, and **faccessx** subroutines allow the specification of a class of users or processes for whom access is to be checked.

The caller must have search permission for all components of the *PathName* parameter.

The **accessxat** subroutine is equivalent to the **accessx** subroutine, and the **faccessat** subroutine is equivalent to the **access** subroutine if the *PathName* parameter specifies an absolute path or if the *DirFileDescriptor* parameter is set to **AT_FDCWD**. The file accessibility is determined by the relative path to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory. If the directory is accessed without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted by using the current permissions of the directory. If the directory is accessed with the **O_SEARCH** open flag, the subroutine does not perform the check.

Parameters

Item	Description
PathName	Specifies the path name of the file. If the *PathName* parameter refers to a symbolic link, the **access** subroutine returns information about the file pointed to by the symbolic link. If the *DirFileDescriptor* is specified and *PathName* is relative, then the *DirFileDescriptor* specifies the effective current working directory for the *PathName*.
FileDescriptor	Specifies the file descriptor of an open file.

Item	Description
Mode	Specifies the access modes to be checked. This parameter is a bit mask containing 0 or more of the following values, which are defined in the **<sys/access.h>** file: **R_OK** Check read permission. **W_OK** Check write permission. **X_OK** Check execute or search permission. **F_OK** Check the existence of a file. If none of these values are specified, the existence of a file is checked.
Who	Specifies the class of users for whom access is to be checked. This parameter must be one of the following values, which are defined in the **<sys/access.h>** file: **ACC_SELF** Determines if access is permitted for the current process. The effective user and group IDs, the concurrent group set and the privilege of the current process are used for the calculation. **ACC_INVOKER** Determines if access is permitted for the invoker of the current process. The real user and group IDs, the concurrent group set, and the privilege of the invoker are used for the calculation. **Note:** The expression **access (***PathName***,** *Mode***)** is equivalent to **accessx (***PathName***,** *Mode***, ACC_INVOKER)**. **ACC_OTHERS** Determines if the specified access is permitted for any user other than the object owner. The *Mode* parameter must contain only one of the valid modes. Privilege is not considered in the calculation. **ACC_ALL** Determines if the specified access is permitted for all users. The *Mode* parameter must contain only one of the valid modes. Privilege is not considered in the calculation . **Note:** The **accessx** subroutine shows the same behavior by both the user and root with **ACC_ALL**.
DirFileDescriptor	Specifies the file descriptor of an open directory, which is used as the effective current working directory for the *PathName* argument. If the *DirFileDescriptor* parameter equals **AT_FDCWD**, the *DirFileDescriptor* parameter is ignored and the *PathName* argument specifies the complete file.
Flag	Specifies a bit field argument. If the *Flag* parameter equals **AT_EACCESS**, the effective user and group IDs are checked (**ACC_SELF**). If the *Flag* parameter is zero, the real IDs are checked (**ACC_INVOKER**).

Return Values

If the requested access is permitted, the **access**, **accessx**, **faccessx**, **accessxat**, and **faccessat** subroutines return a value of 0. If the requested access is not permitted or the function call fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

The **access** subroutine indicates success for **X_OK** even if none of the execute file permission bits are set.

Error Codes

The **access faccessat**, **accessx**, and **accessx** subroutines fail if one or more of the following are true:

Item	Description
EACCES	Search permission is denied on a component of the *PathName* prefix.
EFAULT	The *PathName* parameter points to a location outside the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *PathName* parameter.
ENAMETOOLONG	A component of the *PathName* parameter exceeded 255 characters or the entire *PathName* parameter exceeded 1022 characters.
ENOENT	A component of the *PathName* does not exist or the process has the **disallow truncation** attribute set.
ENOENT	The named file does not exist.
ENOENT	The *PathName* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOTDIR	A component of the *PathName* is not a directory.
ESTALE	The process root or current directory is located in a virtual file system that has been unmounted.

The **faccessx** subroutine fails if the following is true:

Item	Description
EBADF	The value of the *FileDescriptor* parameter is not valid.

The **access**, **accessx**, and **faccessx** subroutines fail if one or more of the following is true:

Item	Description
EACCES	The file protection does not allow the requested access.
ENOMEM	Unable to allocate memory.
EIO	An I/O error occurred during the operation.
EROFS	Write access is requested for a file on a read-only file system.

The **accessxat** and **faccessat** subroutines fail if one or more of the following settings are true:

Item	Description
EBADF	The *PathName* parameter does not specify an absolute path and the *DirFileDescriptor* argument is neither AT_FDCWD nor a valid file descriptor.
EINVAL	The value of the **Flag** parameter is not valid.
ENOTDIR	The *PathName* parameter is not an absolute path and **DirFileDescriptor** is a file descriptor but is not associated with a directory.

If Network File System (NFS) is installed on your system, the **accessx**, **accessxat**, and **faccessx** subroutines can also fail if the following settings are true:

Item	Description
ETIMEDOUT	The connection timed out.

Item	Description
ETXTBSY	Write access is requested for a shared text file that is being executed.
EINVAL	The value of the *Mode* argument is invalid.

accel_compress Subroutine

Purpose

Compresses data by using hardware accelerated memory compression.

Syntax

```
#include <sys/types.h>

#include <sys/vminfo.h>


int accel_compress (void *uc_buf, size_t uc_len,
void *c.buf, size_t *c_lenp, int flags);
```

Description

Given a pointer to a buffer with data to compress, the **accel_compress** subroutine compresses the data into the buffer pointed to by the **c_buf** parameter.

The compression subroutine should be called with the **c_lenp** parameter initialized to the total size of the **c_buf** parameter. Upon successful return, the **c_lenp** parameter is updated with the size of the compressed data in the **c_buf** parameter. The following restrictions apply to this subroutine.

- There is no overlapping of the **uc_buf** parameter and the **c_buf** parameter. An overlap results in an error.
- The **uc_buf** and **c_buf** parameters must be aligned at least on a 128 byte boundary. For the best results, both **uc_buf** and **c_buf** parameters must be aligned on a 4096 byte boundary.
- The **c_len** and ***uc_lenp** parameters are limited to a maximum of 1044480 bytes per subroutine call when buffers are aligned on a 4096 byte boundary. For buffers that are not aligned on a 4096 byte boundary, but are aligned on a 128 byte boundary, the **c_len** and ***uc_lenp** parameters are limited to 1040384 bytes per subroutine call plus any alignment offset from a 4096 byte boundary.
- The **uc_len** and **c_lenp** parameters must be a multiple of 8 bytes.
- The mapping of file segments with the **shmat()** function and the **mmap()** function are not allowed. However, the mapping of non-file segments with the **shmat()** function and the **mmap()** function are allowed (for example, `MMAP_ANONYMOUS`).
- The caller is responsible for supplying a large enough **c_buf**.

The subroutine uses the 842 algorithm to compress the data. The compressed buffer includes a cyclic redundancy check (CRC) which is automatically checked by the **accel_decompress()** subroutine. The Active Memory Expansion (AME) and Active Memory Sharing (AMS) features must not be enabled to use this call. The subroutine supports both 32 and 64 bit applications. The subroutine can be called from either a single or multi-threaded process.

Hardware accelerators are a finite resource on any system and you must be careful to not overwhelm the accelerators. If you have a large pool of threads all competing for a few of the available accelerators, you can end up with worse performance than with pure software compression.

Parameters

Item	Description
uc_buf	Pointer to input buffer with data to compress.
uc_len	Length of data in the **uc_buf** parameter to compress.
c_buf	Pointer to out buffer written with compressed data.
c_lenp	Pointer to in/out parameter. On entry, the **c_lenp** parameter is the total available size in the **c_buf** parameter and on exit, the **c_lenp** parameter is the number of bytes written to the **c_buf** parameter.
flags	Reserved for future use. This parameter must be set to zero.

Execution environment

The **accel_compress** subroutine can be called from the process environment only.

Return Values

Item	Description
0	Success
-1	Error. On failure, the errno global variable is set as follows:

EFAULT
Error accessing memory pointed to by the **c_lenp** parameter or access error on the source or target buffer.

EINVAL
Error due to one of the following conditions:

- The **uc_buf** and **c_buf** parameters have wrong alignment.
- The **uc_buf** and **c_buf** parameter overlap.
- The **uc_len** or **c_lenp** parameter is not a multiple of 8.
- The **uc_buf**, **c_buf,** or **c_lenp** parameter is NULL.
- Failed to create a list of the **uc_buf** or **c_buf** parameter pages to pass on to the accelerator hardware.
- The **uc_buf** or **c_buf** parameters are in a file.
- The **flags** parameter is a nonzero value.

ENOSYS
The hardware accelerator is not available, or AME is enabled, or AMS is enabled.

ENOMEM
Failed to allocate memory inside the subroutine.

EFBIG
The **uc_len** or the **c_lenp** parameter exceed 1,044,480 bytes.

EIO
The firmware call failed or the accelerator hardware returned a failure of unknown type. This might include errors caused by incorrect input arguments to the **accel_compress()** subroutine.

ENOSPC
The **c_buf** parameter is too small to hold the entire compressed output.

ERANGE
The compressed data is larger than the uncompressed data.

accel_decompress Subroutine

Purpose

Decompresses data by using hardware accelerated memory decompression or a slower software decompression if a hardware accelerator is not available.

Syntax

```
#include <sys/types.h>

#include <sys/vminfo.h>


int accel_decompress (void *c_buf, size_t c_len,
void *uc buf, size_t *uc_lenp, int flags);
```

Description

Given a pointer to a buffer with data to decompress (the **c_buf** parameter), the **accel_decompress** subroutine returns the decompressed data in the buffer pointed to by the **uc_buf** parameter.

The compression subroutine should be called with the **uc_lenp** parameter initialized to the total size of the **uc_buf** parameter. Upon successful return, the **uc_lenp** parameter is updated with the size of the compressed data in the **uc_buf** parameter. The following restrictions apply to this subroutine.

- There is no overlapping of the **uc_buf** parameter and the **c_buf** parameter. An overlap results in an error.
- The **uc_buf** and **c_buf** parameters must be aligned at least on a 128 byte boundary. For the best result, both **uc_buf** and **c_buf** parameters must be aligned on a 4096 byte boundary.
- The **c_len** and ***uc_lenp** parameters are limited to a maximum of 1044480 bytes per subroutine call when buffers are aligned on a 4096 byte boundary. For buffers that are not aligned on a 4096 byte boundary, but are aligned on a 128 byte boundary, the **c_len** and ***uc_lenp** parameters are limited to 1040384 bytes per subroutine call plus any alignment offset from a 4096 byte boundary.
- The **uc_lenp** and **c_len** parameters must be a multiple of 8 bytes.
- The mapping of file segments with the **shmat()** function and the **mmap()** function are not allowed. However, the mapping of non-file segments with the **shmat()** function and the **mmap()** function are allowed (for example, `MMAP_ANONYMOUS`).
- The caller is responsible for supplying a large enough **uc_buf**.

The subroutine uses the 842 algorithm to decompress the data. The compressed buffer includes a cyclic redundancy check (CRC) that is added by the **accel_compress()** subroutine, which is verified against the uncompressed data. If an hardware accelerator is not available in the system that is used for decompression, the call uses the software decompression method. The subroutine supports both 32 bit and 64 bit applications.

Hardware accelerators are a finite resource on any system and you must be careful to not overwhelm the accelerators. If you have a large pool of threads all competing for a few of the available accelerators, you can end up with worse performance than with pure software decompression.

Parameters

Item	Description
c_buf	Pointer to input buffer with data to decompress.
c_len	Length of compressed data in the **c_buf** parameter.
uc_buf	Pointer to out buffer written with decompressed data.

Item	Description
uc_lenp	Pointer to in/out parameter. On entry, the **uc_lenp** parameter is the total available size in the **uc_buf** parameter and on exit, the **uc_lenp** parameter is the number of bytes written to the **uc_buf** parameter.
flags	Reserved for future use. This parameter must be set to zero.

Execution environment

The **accel_decompress** subroutine can be called from the process environment only.

Return Values

Item	Description
0	Success
1	Error. On failure, the errno global variable is set as follows: **EFAULT** Error accessing memory pointed to by the **c_lenp** parameter or access error on the source or target buffer. **EINVAL** Error due to one of the following conditions: • The **uc_buf** and **c_buf** parameters have wrong alignment. • The **uc_buf** and **c_buf** parameter overlap. • The **uc_lenp** or **c_len** parameter is not a multiple of 8. • The **uc_buf**, **c_buf,** or **c_lenp** parameter is NULL. • Failed to create a list of the **uc_buf** or **c_buf** parameter pages to pass on to the accelerator hardware. • The **uc_buf** or **c_buf** parameters are in a file. • The **flags** parameter is a nonzero value. **ENOMEM** Failed to allocate memory inside the subroutine. **EFBIG** The **uc_lenp** or the **c_len** parameter exceed 1,044,480 bytes. **EIO** The firmware call failed or the accelerator hardware returned a failure of unknown type. This might include errors caused by incorrect input arguments to the **accel_decompress()** subroutine. **ECORRUPT** The compressed data is invalid or doesn't match embedded CRC. **ENOSPC** The output buffer is too small to hold all decompressed data.

accredrange Subroutine

Purpose

Checks whether the sensitivity label (SL) is in accreditation.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>

int accredrange (sl)
const sl_t *sl;
```

Description

The **accredrange** subroutine checks whether the sensitivity label (SL) is in the accreditation range that the initialized label database defines. The *sl* parameter specifies the sensitivity label to be checked. The label encodings file defines the accreditation range.

Requirement: Must initialize the database before running this subroutine.

Parameter

Item	Description
sl	Specifies the sensitivity label to be checked.

Files Access

Mode	File
r	**/etc/security/enc/LabelEncodings**

Return Values

If the sensitivity label is in the accreditation range, the **accredrange** subroutine returns a value of zero. If the sensitivity label is not in the accreditation range, it returns a value of -1.

Error Codes

If the **accredrange** subroutine fails, it sets one of the following error codes:

Item	Description
EINVAL	The *sl* parameter specifies a sensitivity label that is not valid.
ENOTREADY	The database is not initialized.

acct Subroutine

Purpose

Enables and disables process accounting.

Library

Standard C Library (**libc.a**)

Syntax

```
int acct ( Path)
char *Path;
```

Description

The **acct** subroutine enables the accounting routine when the *Path* parameter specifies the path name of the file to which an accounting record is written for each process that terminates. When the *Path* parameter is a 0 or null value, the **acct** subroutine disables the accounting routine.

If the *Path* parameter refers to a symbolic link, the **acct** subroutine causes records to be written to the file pointed to by the symbolic link.

If Network File System (NFS) is installed on your system, the accounting file can reside on another node.

Note: To ensure accurate accounting, each node must have its own accounting file. Although no two nodes should share accounting files, a node's accounting files can be located on any node in the network.

The calling process must have root user authority to use the **acct** subroutine.

Parameters

Item	Description
Path	Specifies a pointer to the path name of the file or a null pointer.

Return Values

Upon successful completion, the **acct** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **acct** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EACCES	Write permission is denied for the named accounting file.
EACCES	The file named by the *Path* parameter is not an ordinary file.
EBUSY	An attempt is made to enable accounting when it is already enabled.
ENOENT	The file named by the *Path* parameter does not exist.
EPERM	The calling process does not have root user authority.
EROFS	The named file resides on a read-only file system.

If NFS is installed on the system, the **acct** subroutine is unsuccessful if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

acct_wpar Subroutine

Purpose

Enables and disables process accounting.

Syntax

```
int acct_wpar(PathName, flag)
char * PathName;
int flag;
```

Description

The **acct_wpar** subroutine enables the accounting routine when the *PathName* parameter specifies the path name of the file to which an accounting record is written for each process that terminates. When the *PathName* parameter is a 0 or null value, the **acct_wpar** subroutines disables the accounting routine.

The *flag* parameter can be used to indicate whether to include workload partition accounting records into the global workload partition's accounting file.

If Network File System (NFS) is installed on your system, the accounting file can reside on another node.

Note: To ensure accurate accounting, each node must have its own accounting file. Although no two nodes should share accounting files, a node's accounting file can be located on any node in the network.

The calling process must have root user authority to use the **acct_wpar** subroutine.

Parameters

Item	Description
PathName	Specifies a pointer to the path name of the file or a null pointer. If the *PathName* parameter refers to a symbolic link, the **acct_wpar** subroutine causes records to be written to the file pointed to by the symbolic link.
flag	Specifies whether to include workload partition accounting records into the global accounting records file. Valid flags are the following: **ACCT_INC_GLOBAL** Include the global workload partition's accounting records. **ACCT_INC_ALL_WPARS** Include all workload partition's accounting records.

Return Values

Item	Description
0	The command completed successfully.
-1	The command did not complete successfully. The global variable **errno** is set to indicate the error.

Error Codes

Item	Description
EINVAL	Invalid *flag* argument.
EACCES	Write permission is denied for the named accounting file.
EACCES	The file named by the *PathName* parameter is not an ordinary file.
EBUSY	An attempt is made to enable accounting when it is already enabled.
ENOENT	The file named by the *PathName* parameter does not exist.
EPERM	The calling process does not have root user authority.
EROFS	The named file resides on a read-only file system.

If NFS is installed on the system, the **acct_wpar** subroutine is unsuccessful if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

acl_chg or acl_fchg Subroutine

Purpose

Changes the AIXC ACL type access control information on a file.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/access.h>
```

```
int acl_chg (Path, How, Mode, Who)
char * Path;
int  How;
int  Mode;
int  Who;
```

```
int acl_fchg (FileDescriptor, How, Mode, Who)
int  FileDescriptor;
int How;
int Mode;
int Who;
```

Description

The **acl_chg** and **acl_fchg** subroutines modify the AIXC ACL-type-based access control information of a specified file. This call can fail for file system objects with any non-AIXC ACL.

Parameters

Item	Description
FileDescriptor	Specifies the file descriptor of an open file.
How	Specifies how the permissions are to be altered for the affected entries of the Access Control List (ACL). This parameter takes one of the following values: **ACC_PERMIT** Allows the types of access included in the *Mode* parameter. **ACC_DENY** Denies the types of access included in the *Mode* parameter. **ACC_SPECIFY** Grants the access modes included in the *Mode* parameter and restricts the access modes not included in the *Mode* parameter.

Item	Description
Mode	Specifies the access modes to be changed. The *Mode* parameter is a bit mask containing zero or more of the following values: **R_ACC** Allows read permission. **W_ACC** Allows write permission. **X_ACC** Allows execute or search permission.
Path	Specifies a pointer to the path name of a file.
Who	Specifies which entries in the ACL are affected. This parameter takes one of the following values: **ACC_OBJ_OWNER** Changes the owner entry in the base ACL. **ACC_OBJ_GROUP** Changes the group entry in the base ACL. **ACC_OTHERS** Changes all entries in the ACL except the base entry for the owner. **ACC_ALL** Changes all entries in the ACL.

Return Values

On successful completion, the **acl_chg** and **acl_fchg** subroutines return a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **acl_chg** subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOTDIR	A component of the *Path* prefix is not a directory.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

The **acl_fchg** subroutine fails and the file permissions remain unchanged if the following is true:

Item	Description
EBADF	The *FileDescriptor* value is not valid.

The **acl_chg** or **acl_fchg** subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

Item	Description
EINVAL	The *How* parameter is not one of **ACC_PERMIT**, **ACC_DENY**, or **ACC_SPECIFY**.
EINVAL	The *Who* parameter is not **ACC_OWNER**, **ACC_GROUP**, **ACC_OTHERS**, or **ACC_ALL**.
EROFS	The named file resides on a read-only file system.

The **acl_chg** or **acl_fchg** subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

Item	Description
EIO	An I/O error occurred during the operation.
EPERM	The effective user ID does not match the ID of the owner of the file and the invoker does not have root user authority.

If Network File System (NFS) is installed on your system, the **acl_chg** and **acl_fchg** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

acl_get or acl_fget Subroutine

Purpose

Gets the access control information of a file if the ACL associated is of the AIXC type.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/access.h>
```

```
char *acl_get (Path)
char * Path;

char *acl_fget (FileDescriptor)
int  FileDescriptor;
```

Description

The **acl_get** and **acl_fget** subroutines retrieve the access control information for a file system object. This information is returned in a buffer pointed to by the return value. The structure of the data in this buffer is unspecified. The value returned by these subroutines should be used only as an argument to the **acl_put** or **acl_fput** subroutines to copy or restore the access control information. Note that **acl_get** and **acl_fget**

subroutines could fail if the ACL associated with the file system object is of a different type than AIXC. It is recommended that applications make use of **aclx_get** and **aclx_fget** subroutines to retrieve the ACL.

The buffer returned by the **acl_get** and **acl_fget** subroutines is in allocated memory. After usage, the caller should deallocate the buffer using the **free** subroutine.

Parameters

Item	Description
Path	Specifies the path name of the file.
FileDescriptor	Specifies the file descriptor of an open file.

Return Values

On successful completion, the **acl_get** and **acl_fget** subroutines return a pointer to the buffer containing the access control information. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **acl_get** subroutine fails if one or more of the following are true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOTDIR	A component of the *Path* prefix is not a directory.
ENOENT	A component of the *Path* does not exist or the process has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

The **acl_fget** subroutine fails if the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid file descriptor.

The **acl_get** or **acl_fget** subroutine fails if the following is true:

Item	Description
EIO	An I/O error occurred during the operation.

If Network File System (NFS) is installed on your system, the **acl_get** and **acl_fget** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

Security

Item	Description
Access Control	The invoker must have search permission for all components of the *Path* prefix.
Audit Events	None.

acl_put or acl_fput Subroutine

Purpose

Sets AIXC ACL type access control information of a file.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/access.h>
```

```
int acl_put (Path, Access, Free)
char * Path;
char * Access;
int  Free;
```

```
int acl_fput (FileDescriptor, Access, Free)
int  FileDescriptor;
char * Access;
int  Free;
```

Description

The **acl_put** and **acl_fput** subroutines set the access control information of a file system object. This information is contained in a buffer returned by a call to the **acl_get** or **acl_fget** subroutine. The structure of the data in this buffer is unspecified. However, the entire Access Control List (ACL) for a file cannot exceed one memory page (4096 bytes) in size. Note that **acl_put**/**acl_fput** operation could fail if the existing ACL associated with the file system object is of a different kind or if the underlying physical file system does not support AIXC ACL type. It is recommended that applications make use of **aclx_put** and **aclx_fput** subroutines to set the ACL instead of **acl_put**/**acl_fput** routines.

Parameters

Item	Description
Path	Specifies the path name of a file.
FileDescriptor	Specifies the file descriptor of an open file.
Access	Specifies a pointer to the buffer containing the access control information.

Item	Description
Free	Specifies whether the buffer space is to be deallocated. The following values are valid: **0** Space is not deallocated. **1** Space is deallocated.

Return Values

On successful completion, the **acl_put** and **acl_fput** subroutines return a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **acl_put** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOTDIR	A component of the *Path* prefix is not a directory.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

The **acl_fput** subroutine fails and the file permissions remain unchanged if the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid file descriptor.

The **acl_put** or **acl_fput** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

Item	Description
EINVAL	The *Access* parameter does not point to a valid access control buffer.
EINVAL	The *Free* parameter is not 0 or 1.
EIO	An I/O error occurred during the operation.
EROFS	The named file resides on a read-only file system.

If Network File System (NFS) is installed on your system, the **acl_put** and **acl_fput** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the *Path* prefix.

Auditing Events:

Item	Description
Event	**Information**
chacl	*Path*
fchacl	*FileDescriptor*

acl_set or acl_fset Subroutine

Purpose

Sets the AIXC ACL type access control information of a file.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/access.h>

int acl_set (Path, OwnerMode, GroupMode, DefaultMode)
char * Path;
int  OwnerMode;
int  GroupMode;
int  DefaultMode;

int acl_fset (FileDescriptor, OwnerMode, GroupMode, DefaultMode)
int * FileDescriptor;
int OwnerMode;
int GroupMode;
int DefaultMode;
```

Description

The **acl_set** and **acl_fset** subroutines set the base entries of the Access Control List (ACL) of the file. All other entries are discarded. Other access control attributes are left unchanged. Note that if the file system object is associated with any other ACL type access control information, it will be replaced with just the Base mode bits information. It is strongly recommended that applications stop using these interfaces and instead make use of **aclx_put** and **aclx_fput** subroutines to set the ACL.

Parameters

Item	Description
DefaultMode	Specifies the access permissions for the default class.

Item	Description
FileDescriptor	Specifies the file descriptor of an open file.
GroupMode	Specifies the access permissions for the group of the file.
OwnerMode	Specifies the access permissions for the owner of the file.
Path	Specifies a pointer to the path name of a file.

The mode parameters specify the access permissions in a bit mask containing zero or more of the following values:

Item	Description
R_ACC	Authorize read permission.
W_ACC	Authorize write permission.
X_ACC	Authorize execute or search permission.

Return Values

Upon successful completion, the **acl_set** and **acl_fset** subroutines return the value 0. Otherwise, the value -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **acl_set** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOTDIR	A component of the *Path* prefix is not a directory.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

The **acl_fset** subroutine fails and the file permissions remain unchanged if the following is true:

Item	Description
EBADF	The file descriptor *FileDescriptor* is not valid.

The **acl_set** or **acl_fset** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

Item	Description
EIO	An I/O error occurred during the operation.
EPERM	The effective user ID does not match the ID of the owner of the file and the invoker does not have root user authority.
EROFS	The named file resides on a read-only file system.

If Network File System (NFS) is installed on your system, the **acl_set** and **acl_fset** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the *Path* prefix.

Auditing Events:

Event	Information
chacl	*Path*
fchacl	*FileDescriptor*

aclx_convert Subroutine

Purpose

Converts the access control information from one ACL type to another.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/acl.h>
```

```
int aclx_convert (from_acl, from_sz, from_type, to_acl, to_sz, to_type, fs_obj_path)
void * from_acl;
size_t   from_sz;
acl_type_t   from_type;
void * to_acl;
size_t  * to_sz;
acl_type_t   to_type;
char  * fs_obj_path;
```

Description

The **aclx_convert** subroutine converts the access control information from the binary input given in *from_acl* of the ACL type *from_type* into a binary ACL of the type *to_type* and stores it in *to_acl*. Values *from_type* and *to_type* can be any ACL types supported in the system.

The ACL conversion takes place with the help of an ACL type-specific algorithm. Because the conversion is approximate, it can result in a potential loss of access control. Therefore, the user of this call must make sure that the converted ACL satisfies the required access controls. The user can manually review

the access control information after the conversion for the file system object to ensure that the conversion was successful and satisfied the requirements of the intended access control.

Parameters

Item	Description
from_acl	Points to the ACL that has to be converted.
from_sz	Indicates the size of the ACL information pointed to by *from_acl*.
from_type	Indicates the ACL type information of the ACL. The *acl_type* is 64 bits in size and is unique on the system. If the given *acl_type* is not supported in the system, this function fails and **errno** is set to **EINVAL**. The supported ACL types are **ACLX** and **NFS4**.
to_acl	Points to a buffer in which the target binary ACL has to be stored. The amount of memory available in this buffer is indicated by the *to_sz* parameter.
to_sz	Indicates the amount of memory, in bytes, available in *to_acl*. If *to_sz* contains less than the required amount of memory for storing the converted ACL, **to_sz* is set to the required amount of memory and **ENOSPC** is returned by **errno**.
to_type	Indicates the ACL type to which conversion needs to be done. The ACL type is 64 bits in size and is unique on the system. If the given *acl_type* is not supported in the system, this function fails and **errno** is set to **EINVAL** The supported ACL types are **ACLX** and **NFS4**.
fs_obj_path	File System Object Path for which the ACL conversion is being requested. Gets information about the object, such as whether it is file or directory.

Return Values

On successful completion, the **aclx_convert** subroutine returns a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **aclx_convert** subroutine fails if one or more of the following is true:

Item	Description
EINVAL	Invalid input parameter. The same error can be returned if an invalid *acl_type* is specified as input to this routine, either in *from_type* or in *to_type*. This **errno** could also be returned if the binary ACL given in *from_acl* is not the type specified by *from_type*.
ENOSPC	Insufficient storage space is available in *to_acl*.

Security

Access Control: The invoker must have search permission for all components of the *Path* prefix.

Auditing Events: If the auditing subsystem has been properly configured and is enabled, the **aclx_convert** subroutine generates the following audit record (event) every time the command is executed:

Item	Description
Event	**Information**
FILE_Acl	Lists access controls.

aclx_get or aclx_fget Subroutine

Purpose

Gets the access control information for a file system object.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/acl.h>
```

```
int aclx_get (Path, ctl_flags, acl_type, acl, acl_sz,  mode_info)
char * Path;
uint64_t   ctl_flags;
acl_type_t  * acl_type;
void * acl;
size_t  * acl_sz;
mode_t  * mode_info;
```

```
int aclx_fget (FileDescriptor, ctl_flags, acl_type, acl, acl_sz,  mode_info)
int  FileDescriptor;
uint64_t   ctl_flags;
acl_type_t  * acl_type;
void * acl;
size_t  * acl_sz;
mode_t  * mode_info;
```

Description

The **aclx_get** and **aclx_fget** subroutines retrieve the access control information for a file system object in the native ACL format. Native ACL format is the format as defined for the particular ACL type in the system. These subroutines are advanced versions of the **acl_get** and **acl_fget** subroutines and should be used instead of the older versions. The **aclx_get** and **aclx_fget** subroutines provide for more control for the user to interact with the underlying file system directly.

In the earlier versions (**acl_get** or **acl_fget**), OS libraries found out the ACL size from the file system and allocated the required memory buffer space to hold the ACL information. The caller does all this now with the **aclx_get** and **aclx_fget** subroutines. Callers are responsible for finding out the size and allocating memory for the ACL information, and later freeing the same memory after it is used. These subroutines allow for an *acl_type* input and output argument. The data specified in this argument can be set to a particular ACL type and a request for the ACL on the file system object of the same type. Some physical file systems might do emulation to return the ACL type requested, if the ACL type that exists on the file system object is different. If the *acl_type* pointer points to a data area with a value of **ACL_ANY** or 0, then the underlying physical file system has to return the type of the ACL associated with the file system object.

The *ctl_flags* parameter is a bit mask that allows for control over the **aclx_get** requests.

The value returned by these subroutines can be use as an argument to the **aclx_get** or **aclx_fget** subroutines to copy or restore the access control information.

Parameters

Item	Description
Path	Specifies the path name of the file system object.
FileDescriptor	Specifies the file descriptor of an open file.

Item	Description
ctl_flags	This 64-bit sized bit mask provides control over the ACL retrieval. The following flag value is defined: **GET_ACLINFO_ONLY** Gets only the ACL type and length information from the underlying file system. When this bit is set, the *acl* argument can be set to NULL. In all other cases, these must be valid buffer pointers (or else an error is returned). If this bit is not specified, then all the other information about the ACL, such as ACL data and mode information, is returned.
acl_type	Points to a buffer that will hold ACL type information. The ACL type is 64 bits in size and is unique on the system. The caller can provide an ACL type in this area and a request for the ACL on the file system object of the same type. If the ACL type requested does not match the one on the file system object, the physical file system might return an error or emulate and provide the ACL information in the ACL type format requested. If the caller does not know the ACL type and wants to retrieve the ACL associated with the file system object, then the caller should set the buffer value pointed to by *acl_type* to **ACL_ANY** or 0. The supported ACL types are **ACLX** and **NFS4**.
acl	Points to a buffer where the ACL retrieved is stored. The size of this buffer is indicated by the *acl_sz* parameter.
acl_sz	Indicates the size of the buffer area passed through the *acl* parameter.
mode_info	Pointer to a buffer where the mode word associated with the file system object is returned. Note that this mode word's meaning and formations depend entirely on the ACL type concerned.

Return Values

On successful completion, the **aclx_get** and **aclx_get** subroutines return a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **aclx_get** subroutine fails if one or more of the following is true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOTDIR	A component of the *Path* prefix is not a directory.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

The **aclx_fget** subroutine fails if the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid file descriptor.

The **aclx_get** or **aclx_fget** subroutine fails if one or more of the following is true:

Item	Description
EINVAL	Invalid input parameter. The same error can be returned if an invalid *acl_type* is specified as input to this routine.
EIO	An I/O error occurred during the operation.
ENOSPC	Input buffer size *acl_sz* is not sufficient to store the ACL data in *acl*.

If Network File System (NFS) is installed on your system, the **aclx_get** and **aclx_fget** subroutines can also fail if the following condition is true:

Item	Description
ETIMEDOUT	The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the *Path* prefix.

Auditing Events: None

aclx_gettypeinfo Subroutine

Purpose

Retrieves the ACL characteristics given to an ACL type.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/acl.h>

int aclx_gettypeinfo (Path, acl_type, buffer, buffer_sz)
char * Path;
acl_type_t   acl_type;
caddr_t  buffer;
size_t  * buffer_sz;
```

Description

The **aclx_gettypeinfo** subroutine helps obtain characteristics and capabilities of an ACL type on the file system. The buffer space provided by the caller is where the ACL type-related information is returned. If the length of this buffer is not enough to fit the characteristics for the ACL type requested, then **aclx_gettypeinfo** returns an error and sets the *buffer_len* field to the amount of buffer space needed.

Parameters

Item	Description
Path	Specifies the path name of the file.
acl_type	ACL type for which the characteristics are sought. The supported ACL types are **ACLX** and **NFS4**.
buffer	Specifies the pointer to a buffer space, where the characteristics of *acl_type* for the file system is returned. The structure of data returned is ACL type-specific. Refer to the ACL type-specific documentation for more details.
buffer_sz	Points to an area that specifies the length of the buffer *buffer* in which the characteristics of *acl_type* are returned by the file system. This is an input/output parameter. If the length of the buffer provided is not sufficient to store all the ACL type characteristic information, then the file system returns an error and indicates the length of the buffer required in this variable. The length is specified in number of bytes.

Return Values

On successful completion, the **aclx_gettypeinfo** subroutine returns a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **aclx_gettypeinfo** subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOSPC	Buffer space provided is not enough to store all the *acl_type* characteristics of the file system.
ENOTDIR	A component of the *Path* prefix is not a directory.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

If Network File System (NFS) is installed on your system, the **acl_gettypeinfo** subroutine can also fail if the following condition is true:

Item	Description
ETIMEDOUT	The connection timed out.

Security

Auditing Events: None

aclx_gettypes Subroutine

Purpose

Retrieves the list of ACL types supported for the file system associated with the path provided.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/acl.h>

int aclx_gettypes (Path, acl_type_list, acl_type_list_len)
char  * Path;
acl_types_list_t  * acl_type_list;
size_t  * acl_type_list_len;
```

Description

The **aclx_gettypes** subroutine helps obtain the list of ACL types supported on the particular file system. A file system can implement policies to support one to many ACL types simultaneously. The first ACL type in the list is the default ACL type for the file system. This default ACL type is used in ACL conversions if the target ACL type is not supported on the file system. Each file system object in the file system is associated with only one piece of ACL data of a particular ACL type.

Parameters

Item	Description
Path	Specifies the path name of the file system object within the file system for which the list of supported ACLs are being requested.
acl_type_list	Specifies the pointer to a buffer space, where the list of ACL types is returned. The size of this buffer is indicated using the *acl_type_list_len* argument in bytes. The supported ACL types are **ACLX** and **NFS4**.
acl_type_list_len	Pointer to a buffer that specifies the length of the buffer *acl_type_list* in which the list of ACLs is returned by the file system. This is an input/output parameter. If the length of the buffer is not sufficient to store all the ACL types, the file system returns an error and indicates the length of the buffer required in this same area. The length is specified in bytes. If the subroutine call is successful, this field contains the number of bytes of information stored in the *acl_type_list* buffer. This information can be used by the caller to get the number of ACL type entries returned.

Return Values

On successful completion, the **aclx_gettypes** subroutine returns a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **aclx_gettypes** subroutine fails and the access control information for a file remains unchanged if one or more of the following is true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOSPC	The *acl_type_list* buffer provided is not enough to store all the ACL types supported by this file system.
ENOTDIR	A component of the *Path* prefix is not a directory.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

If Network File System (NFS) is installed on your system, the **acl_gettypes** subroutine can also fail if the following condition is true:

Item	Description
ETIMEDOUT	The connection timed out.

Security

Access Control: Caller must have search permission for all components of the *Path* prefix.

Auditing Events: None

aclx_print or aclx_printStr Subroutine

Purpose

Converts the binary access control information into nonbinary, readable format.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/acl.h>

int aclx_print (acl_file, acl, acl_sz, acl_type, fs_obj_path, flags)
FILE * acl_file;
void  * acl;
```

```
size_t    acl_sz;
acl_type_t   acl_type;
char  * fs_obj_path;
int32_t    flags;

int aclx_printStr (str, str_sz, acl, acl_sz, acl_type, fs_obj_path, flags)
char * str;
size_t * str_sz;
void  * acl;
size_t    acl_sz;
acl_type_t   acl_type;
char  * fs_obj_path;
int32_t    flags;
```

Description

The **aclx_print** and **aclx_printStr** subroutines print the access control information in a nonbinary, readable text format. These subroutines take the ACL information in binary format as input, convert it into text format, and print that text format output to either a file or a string. The **aclx_print** subroutine prints the ACL text to the file specified by *acl_file*. The **aclx_printStr** subroutine prints the ACL text to *str*. The amount of space available in *str* is specified in *str_sz*. If this memory is insufficient, the subroutine sets *str_sz* to the needed amount of memory and returns an **ENOSPC** error.

Parameters

Item	Description
acl_file	Points to the file into which the textual output is printed.
str	Points to the string into which the textual output should be printed.
str_sz	Indicates the amount of memory in bytes available in *str*. If the text representation of *acl* requires more space than *str_sz*, this subroutine updates the *str_sz* with the amount of memory required and fails by setting **errno** to **ENOSPC**.
acl	Points to a buffer which contains the binary ACL data that has to be printed. The size of this buffer is indicated by the *acl_sz* parameter.
acl_sz	Indicates the size of the buffer area passed through the *acl* parameter.
acl_type	Indicates the ACL type information of the *acl*. The ACL type is 64 bits in size and is unique on the system. If the given ACL type is not supported in the system, this function fails and **errno** is set to **EINVAL**. The supported ACL types are **ACLX** and **NFS4**.
fs_obj_path	File System Object Path for which the ACL data format and print are being requested. Gets information about the object (such as whether the object is a file or directory, who the owner is, and the associated group ID).
flags	Allows for control over the print operation. A value of **ACL_VERBOSE** indicates whether additional information has to be printed in text format in comments. This bit is set when the **aclget** command is issued with the **-v** (verbose) option.

Return Values

On successful completion, the **aclx_print** and **aclx_printStr** subroutines return a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **aclx_print** subroutine fails if one or more of the following is true:

Note: The errors in the following list occur only because **aclx_print** calls the **fprintf** subroutine internally. For more information about these errors, refer to the **fprintf** subroutine.

Item	Description
EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying the file specified by the *acl_file* parameter, and the process would be delayed in the write operation.
EBADF	The file descriptor underlying the file specified by the *acl_file* parameter is not a valid file descriptor open for writing.
EFBIG	An attempt was made to write to a file that exceeds the file size limit of this process or the maximum file size. For more information, refer to the **ulimit** subroutine.
EINTR	The write operation terminated because of a signal was received, and either no data was transferred or a partial transfer was not reported.
EIO	The process is a member of a background process group attempting to perform a write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process has no parent process.
ENOSPC	No free space remains on the device that contains the file.
ENOSPC	Insufficient storage space is available.
ENXIO	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
EPIPE	An attempt was made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal is sent to the process.

The **aclx_printStr** subroutine fails if the following is true:

Item	Description
ENOSPC	Input buffer size *strSz* is not sufficient to store the text representation of *acl* in *str*.
ENOSPC	Insufficient storage space is available. This error is returned by **sprintf**, which is called by the **aclx_printStr** subroutine internally.

The **aclx_print** or **aclx_printStr** subroutine fails if the following is true:

Item	Description
EINVAL	Invalid input parameter. The same error can be returned if an invalid *acl_type* is specified as input to this routine. This **errno** can also be returned if the *acl* is not of the type specified by *acl_type*.

aclx_put or aclx_fput Subroutine

Purpose

Stores the access control information for a file system object.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/acl.h>
```

```
int aclx_put (Path, ctl_flags, acl_type, acl, acl_sz,  mode_info)
char * Path;
uint64_t   ctl_flags;
acl_type_t   acl_type;
void * acl;
size_t   acl_sz;
mode_t   mode_info;
```

```
int aclx_fput (FileDescriptor, ctl_flags, acl_type, acl, acl_sz,  mode_info)
int  FileDescriptor;
uint64_t   ctl_flags;
acl_type_t   acl_type;
void * acl;
size_t   acl_sz;
mode_t   mode_info;
```

Description

The **aclx_put** and **aclx_fput** subroutines store the access control information for a file system object in the native ACL format. Native ACL format is the format as defined for the particular ACL type in the system. These subroutines are advanced versions of the **acl_put** and **acl_fput** subroutines and should be used instead of the older versions. The **aclx_put** and **aclx_fput** subroutines provide for more control for the user to interact with the underlying file system directly.

A caller specifies the ACL type in the *acl_type* argument and passes the ACL information in the *acl* argument. The *acl_sz* parameter indicates the size of the ACL data. The *ctl_flags* parameter is a bitmask that allows for variation of **aclx_put** requests.

The value provided to these subroutines can be obtained by invoking **aclx_get** or **aclx_fget** subroutines to copy or restore the access control information.

The **aclx_put** and **aclx_fput** subroutines can also be used to manage the special bits (such as SGID and SUID) in the mode word associated with the file system object. For example, you can set the **mode_info** value to any special bit mask (as in the mode word defined for the file system), and a request can be made to set the same bits using the *ctl_flags* argument. Note that special privileges (such as root) might be required to set these bits.

Parameters

Item	Description
Path	Specifies the path name of the file system object.
FileDescriptor	Specifies the file descriptor of an open file system object. This 64-bit sized bit mask provides control over the ACL retrieval. These bits are divided as follows: **Lower 16 bits** System-wide (nonphysical file-system-specific) ACL control flags **32 bits** Reserved. **Last 16 bits** Any physical file-system-defined options (that are specific to physical file system ACL implementation).

Item	Description
ctl_flags	Bit mask with the following system-wide flag values defined: **SET_MODE_S_BITS** Indicates that the **mode_info** value is set by the caller and the ACL put operation needs to consider this value while completing the ACL put operation. **SET_ACL** Indicates that the *acl* argument points to valid ACL data that needs to be considered while the ACL put operation is being performed. **Note:** Both of the preceding values can be specified by the caller by ORing the two masks.
acl_type	Indicates the type of ACL to be associated with the file object. If the *acl_type* specified is not among the ACL types supported for the file system, then an error is returned. The supported ACL types are **ACLX** and **NFS4**.
acl	Points to a buffer where the ACL information exists. This ACL information is associated with the file system object specified. The size of this buffer is indicated by the *acl_sz* parameter.
acl_sz	Indicates the size of the ACL information sent through the *acl* parameter.
mode_info	This value indicates any mode word information that needs to be set for the file system object in question as part of this ACL put operation. When mode bits are being altered by specifying the **SET_MODE_S_BITS** flag (in *ctl_flags*) ACL put operation fails if the caller does not have the required privileges.

Return Values

On successful completion, the **aclx_put** and **aclx_fput** subroutines return a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **aclx_put** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOTDIR	A component of the *Path* prefix is not a directory.

Item	Description
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.

The **aclx_fput** subroutine fails and the file permissions remain unchanged if the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid file descriptor.

The **aclx_put** or **aclx_fput** subroutine fails if one or more of the following is true:

Item	Description
EINVAL	Invalid input parameter. The same error can be returned if an invalid *acl_type* is specified as input to this routine.
EIO	An I/O error occurred during the operation.
EROFS	The named file resides on a read-only file system.

If Network File System (NFS) is installed on your system, the **acl_put** and **acl_fput** subroutines can also fail if the following condition is true:

Item	Description
ETIMEDOUT	The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the *Path* prefix.

Auditing Events:

Item	Description
Event	**Information**
chacl	*Path*-based event
fchacl	*FileDescriptor*-based event

aclx_scan or aclx_scanStr Subroutine

Purpose

Reads the access control information that is in nonbinary, readable text format, and converts it into ACL type-specific native format binary ACL data.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/acl.h>

int aclx_scan (acl_file, acl, acl_sz, acl_type, err_file)
FILE * acl_file;
void  * acl;
size_t   * acl_sz;
```

```
acl_type_t   acl_type;
FILE  * err_file;

int aclx_scanStr (str, acl, acl_sz, acl_type)
char * str;
void  * acl;
size_t   * acl_sz;
acl_type_t   acl_type;
```

Description

The **aclx_scan** and **aclx_scanStr** subroutines read the access control information from the input given in nonbinary, readable text format and return a binary ACL data in the ACL type-specific native format. The **aclx_scan** subroutine provides the ACL data text in the file specified by *acl_file*. In the case of **aclx_scanStr**, the ACL data text is provided in the string pointed to by *str*. When the *err_file* parameter is not Null, it points to a file to which any error messages are written out by the **aclx_scan** subroutine in case of syntax errors in the input ACL data. The errors can occur if the syntax of the input text data does not adhere to the required ACL type-specific data specifications.

Parameters

Item	Description
acl_file	Points to the file from which the ACL text output is read.
str	Points to the string from which the ACL text output is printed.
acl	Points to a buffer in which the binary ACL data has to be stored. The amount of memory available in this buffer is indicated by the *acl_sz* parameter.
acl_sz	Indicates the amount of memory, in bytes, available in the *acl* parameter.
acl_type	Indicates the ACL type information of the *acl*. The ACL type is 64 bits in size and is unique on the system. If the given ACL type is not supported in the system, this function fails and **errno** is set to **EINVAL**. The supported ACL types are **ACLX** and **NFS4**.
err_file	File pointer to an error file. When this pointer is supplied, the subroutines write out any errors in the syntax/composition of the ACL input data.

Return Values

On successful completion, the **aclx_scan** and **aclx_scanStr** subroutines return a value of 0. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **aclx_scan** subroutine fails if one or more of the following is true:

Note: The errors in the following list occur only because **aclx_scan** calls the **fscanf** subroutine internally. For more information about these errors, refer to the **fscanf** subroutine.

Item	Description
EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying the file specified by the *acl_file* parameter, and the process would be delayed in the write operation.
EBADF	The file descriptor underlying the file specified by the *acl_file* parameter is not a valid file descriptor open for writing.

Item	Description
EINTR	The write operation terminated because of a signal was received, and either no data was transferred or a partial transfer was not reported.
EIO	The process is a member of a background process group attempting to perform a write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process has no parent process.
ENOSPC	Insufficient storage space is available.

The **aclx_scanStr** subroutine fails if the following is true:

Item	Description
ENOSPC	Insufficient storage space is available. This error is returned by **sscanf**, which is called by the **aclx_scanStr** subroutine internally.

The **aclx_scan** or **aclx_scanStr** subroutine fails if the following is true:

Item	Description
EINVAL	Invalid input parameter. The same error can be returned if an invalid *acl_type* is specified as input to this routine. This **errno** can also be returned if the text ACL given in the input/file string is not of the type specified by *acl_type*.

acos, acosf, acosl, acosd32, acosd64, or acosd128 Subroutines

Purpose

Computes the inverse cosine of a given value.

Syntax

```
#include <math.h>

float acosf (x)
float x;

long double acosl (x)
long double x;

double acos (x)
double x;
_Decimal32 acosd32 (x)
_Decimal32 x;

_Decimal64 acosd64 (x)
_Decimal64 x;

_Decimal128 acosd128 (x)
_Decimal128 x;
```

Description

The **acosf**, **acosl**, **acos**, **acosd32**, **acosd64**, and **acosd128** subroutines compute the principal value of the arc cosine of the *x* parameter. The value of *x* should be in the range [-1,1].

An application wishing to check for error situations should set the **errno** global variable to zero and call **fetestexcept(FE_ALL_EXCEPT)** before calling these functions. On return, if **errno** is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, these subroutines return the arc cosine of *x*, in the range [0, pi] radians.

For finite values of *x* not in the range [-1,1], a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is +1, 0 is returned.

If *x* is ±Inf, a domain error occurs, and a NaN is returned.

acosh, acoshf, acoshl, acoshd32, acoshd64, and acoshd128 Subroutines

Purpose

Computes the inverse hyperbolic cosine.

Syntax

```
#include <math.h>

float acoshf (x)
float x;

long double acoshl (X)
long double x;

double acosh (x)
double x;
_Decimal32 acoshd32 (x)
_Decimal32 x;

_Decimal64 acoshd64 (x)
_Decimal64 x;

_Decimal128 acoshd128 (x)
_Decimal128 x;
```

Description

The **acoshf**, **acoshl**, **acoshd32**, **acoshd64**, and **acoshd128** subroutines compute the inverse hyperbolic cosine of the *x* parameter.

The **acosh** subroutine returns the hyperbolic arc cosine specified by the *x* parameter, in the range 1 to the **+HUGE_VAL** value.

An application wishing to check for error situations should set **errno** to zero and call **fetestexcept(FE_ALL_EXCEPT)** before calling these subroutines. Upon return, if the **errno** global variable is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **acoshf**, **acoshl**, **acoshd32**, **acoshd64**, and **acoshd128** subroutines return the inverse hyperbolic cosine of the given argument.

For finite values of *x* < 1, a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is +1, 0 is returned.

If *x* is +Inf, +Inf is returned.

If *x* is -Inf, a domain error occurs, and a NaN is returned.

Error Codes

The **acosh** subroutine returns **NaNQ** (not-a-number) and sets **errno** to **EDOM** if the *x* parameter is less than the value of 1.

addch, mvaddch, mvwaddch, or waddch Subroutine

Purpose

Adds a single-byte character and rendition to a window and advances the cursor.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int addch(const chtype ch);
```

```
int mvaddch(int y,
int x,
const chtype ch);
```

```
int mvwaddch(WINDOW *in,
const chtype ch);
```

```
int waddch(WINDOW *win,
const chtype ch);
```

Description

The **addch**, **waddch**, **mvaddch**, and **mvwaddch** subroutines add a character to a window at the logical cursor location. After adding the character, curses advances the position of the cursor one character. At the right margin, an automatic new line is performed.

The **addch** subroutine adds the character to the stdscr at the current logical cursor location. To add a character to a user-defined window, use the **waddch** and **mvwaddch** subroutines. The **mvaddch** and **mvwaddch** subroutines move the logical cursor before adding a character.

If you add a character to the bottom of a scrolling region, curses automatically scrolls the region up one line from the bottom of the scrolling region if **scrollok** is enabled. If the character to add is a tab, new-line, or backspace character, curses moves the cursor appropriately in the window to reflect the addition. Tabs are set at every eighth column. If the character is a new-line, curses first uses the **wclrtoeol** subroutine to erase the current line from the logical cursor position to the end of the line before moving the cursor.

You can also use the **addch** subroutines to add control characters to a window. Control characters are drawn in the ^X notation.

Adding Video Attributes and Text

Because the *Char* parameter is an integer, not a character, you can combine video attributes with a character by ORing them into the parameter. The video attributes are also set. With this capability you can copy text and video attributes from one location to another using the **inch** (inch) and **addch** subroutines.

Parameters

Item	Description
ch	
y	
x	
**win*	

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To add the character *H* represented by variable *x* to stdscr at the current cursor location, enter:

```
chtype x;
x='H';
addch(x);
```

2. To add the x character to stdscr at the coordinates y = 10, x = 5, enter:

```
mvaddch(10, 5, 'x');
```

3. To add the x character to the user-defined window `my_window` at the coordinates y = 10, x = 5, enter:

```
WINDOW *my_window;
mvwaddch(my_window, 10, 5, 'x');
```

4. To add the x character to the user-defined window `my_window` at the current cursor location, enter:

```
WINDOW *my_window;
waddch(my_window, 'x');
```

5. To add the character *x* in standout mode, enter:

```
waddch(my_window, 'x' | A_STANDOUT);
```

This allows 'x' to be highlighted, but leaves the rest of the window alone.

addnstr, addstr, mvaddnstr, mvaddstr, mvwaddnstr, mvwaddstr, waddnstr, or waddstr Subroutine

Purpose

Adds a string of multi-byte characters without rendition to a window and advances the cursor.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int addnstr(const char *str,
int n);
```

```
int addstr(const char *str);
```

```
int mvaddnstr(int y,
int x,
const char *str,
int n);
```

```
int mvaddstr(int y,
int x,
const char *str);
```

```
int mvwaddnstr(WINDOW *win,
int y,
int x,
const char *str,
int n);
```

```
int mvwaddstr(WINDOW *win,
int y,
int x,
const char *str);
```

```
int waddnstr(WINDOW *win,
const char *str,
int n);
```

```
int waddstr(WINDOW *win,
const char *str);
```

Description

These subroutines write the characters of the string *str* on the current or specified window starting at the current or specified position using the background rendition.

These subroutines advance the cursor position, perform special character processing, and perform wrapping.

The **addstr**, **mvaddstr**, **mvwaddstr** and **waddstr** subroutines are similar to calling **mbstowcs** on *str*, and then calling **addwstr**, **mvaddwstr**, **mvwaddwstr**, and **waddwstr**, respectively.

The **addnstr**, **mvaddnstr**, **mvwaddnstr** and **waddnstr** subroutines use at most, *n* bytes from *str*. These subroutines add the entire string when *n* is -1.

Parameters

Item	Description
Column	Specifies the horizontal position to move the cursor to before adding the string.
Line	Specifies the vertical position to move the cursor to before adding the string.
String	Specifies the string to add.
Window	Specifies the window to add the string to.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To add the string represented by `xyz` to the stdscr at the current cursor location, enter:

```
char *xyz;
xyz="Hello!";
addstr(xyz);
```

2. To add the `"Hit a Key"` string to the stdscr at the coordinates y=10, x=5, enter:

```
mvaddstr(10, 5, "Hit a Key");
```

3. To add the `xyz` string to the user-defined window `my_window` at the coordinates y=10, x=5, enter:

```
mvwaddstr(my_window, 10, 5, "xyz");
```

4. To add the `xyz` string to the user-defined string at the current cursor location, enter:

```
waddstr(my_window, "xyz");
```

addproj Subroutine

Purpose

Adds an API-based project definition to the kernel project registry.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

addproj(struct project *)
```

Description

The **addproj** subroutine defines the application-based project definition to the kernel repository. An application can assign a project defined in this way using the `proj_execve` system call.

Projects that are added this way are marked as being specified by applications so that they do not overlap with system administrator-specified projects defined using the `projctl` command. The `PROJFLAG_API` flag is turned on in the structure project to indicate that the project definition was added by an application.

Projects added by a system administrator using the `projctl` command are flagged as being derived from the local or LDAP-based project repositories by the `PROJFLAGS_LDAP` or `PROJFLAGS_PDF` flag. If one of these flags is specified, the `addproj` subroutine fails with `EPERM`.

The `getproj` routine can be used to determine the origin of a loaded project.

The **addproj** validates the input project number to ensure that it is within the expected range of 0x00000001 - 0x00ffffff. It also validates that the project name is a POSIX compliant alphanumeric character string. If any invalid input is found errno will be set to **EINVAL** and the **addproj** subroutine returns -1.

Parameters

Item	Description
project	Points to a project structure that holds the definition of the project to be added.

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Invalid Project Name / Number or the passed pointer is NULL
EEXIST	Project Definition exists
EPERM	Permission Denied, not a privileged user

addprojdb Subroutine

Purpose

Adds a project definition to the specified project database.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

addprojdb(void *handle, struct project *project, char *comment)
```

Description

The **addprojdb** subroutine appends the project definition stored in the struct *project* variable into the project database named by the *handle* parameter. The project database must be initialized before calling this subroutine. The **projdballoc** subroutine is provided for this purpose. This routine verifies whether the supplied project definition already exists. If it does exist, the **addprojdb** subroutine sets errno to **EEXIST** and returns -1.

The **addprojdb** subroutine validates the input project number to ensure that it is within the expected range 0x00000001 - 0x00ffffff and validates that the project name is a POSIX-compliant alphanumeric character string. If any invalid input is found, the **addprojdb** subroutine sets errno to **EINVAL** and returns -1.

If the user does not have privilege to add an entry to project database, the **addprojdb** subroutine sets errno to **EACCES** and returns -1.

There is an internal state (that is, the current project) associated with the project database. When the project database is initialized, the current project is the first project in the database. The **addprojdb** subroutine appends the specified project to the end of the database. It advances the current project assignment to the next project in the database, which is the end of the project data base. At this point, a call to the **getnextprojdb** subroutine would fail, because there are no additional project definitions. To read the project definition that was just added, use the **getprojdb** subroutine. To read other projects, first call **getfirstprojdb** subroutine to reset the internal current project assignment so that subsequent reads can be performed.

The format of the records added to the project database are given as follows:

```
ProjectName:ProjectNumber:AggregationStatus:Comment::
```

Example:

```
Biology:4756:no:Project Created by projctl command::
```

Parameters

Item	Description
handle	Pointer to project database handle
project	Pointer to a project structure that holds the definition of the project to be added
comment	Pointer to a character string that holds the comments about the project

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Invalid project name or number, or the passed pointer is NULL.
EEXIST	Project definition already exists.
EPERM	Permission denied. The user is not a privileged user.

addssys Subroutine

Purpose

Adds the **SRCsubsys** record to the subsystem object class.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
```

int addssys (*SRCSubsystem*)
**struct SRCsubsys ***SRCSubsystem*;

Description

The **addssys** subroutine adds a record to the subsystem object class. You must call the **defssys** subroutine to initialize the *SRCSubsystem* buffer before your application program uses the **SRCsubsys** structure. The **SRCsubsys** structure is defined in the **/usr/include/sys/srcobj.h** file.

The executable running with this subroutine must be running with the group system.

Parameters

Item	Description
SRCSubsystem	A pointer to the **SRCsubsys** structure.

Return Values

Upon successful completion, the **addssys** subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **odmerrno** variable is set to indicate the error, or an SRC error code is returned.

Error Codes

The **addssys** subroutine fails if one or more of the following are true:

Item	Description
SRC_BADFSIG	Invalid stop force signal.
SRC_BADNSIG	Invalid stop normal signal.
SRC_CMDARG2BIG	Command arguments too long.
SRC_GRPNAM2BIG	Group name too long.
SRC_NOCONTACT	Contact not signal, sockets, or message queue.
SRC_NONAME	No subsystem name specified.
SRC_NOPATH	No subsystem path specified.
SRC_PATH2BIG	Subsystem path too long.
SRC_STDERR2BIG	stderr path too long.
SRC_STDIN2BIG	stdin path too long.
SRC_STDOUT2BIG	stdout path too long.
SRC_SUBEXIST	New subsystem name already on file.
SRC_SUBSYS2BIG	Subsystem name too long.
SRC_SYNEXIST	New subsystem synonym name already on file.
SRC_SYN2BIG	Synonym name too long.

Security

Privilege Control: This command has the Trusted Path attribute. It has the following kernel privilege:

Item	Description
SET_PROC_AUDIT	
Files Accessed:	

Mode	File
644	**/etc/objrepos/SRCsubsys**

Auditing Events:

If the auditing subsystem has been properly configured and is enabled, the **addssys** subroutine generates the following audit record (event) each time the subroutine is executed:

Event	Information
SRC_addssys	Lists the SRCsubsys records added.

Files

Item	Description
/etc/objrepos/SRCsubsys	SRC Subsystem Configuration object class.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.
/usr/include/spc.h	Defines external interfaces provided by the SRC subroutines.
/usr/include/sys/srcobj.h	Defines object structures used by the SRC.

adjtime Subroutine

Purpose

Corrects the time to allow synchronization of the system clock.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>
int adjtime ( Delta,  Olddelta)
struct timeval *Delta;
struct timeval *Olddelta;
```

Description

The **adjtime** subroutine makes small adjustments to the system time, as returned by the **gettimeofday** subroutine, advancing or retarding it by the time specified by the *Delta* parameter of the **timeval** structure. If the *Delta* parameter is negative, the clock is slowed down by periodically subtracting a small amount from it until the correction is complete. If the *Delta* parameter is positive, a small amount is periodically added to the clock until the correction is complete. The skew used to perform the correction is generally ten percent. If the clock is sampled frequently enough, an application program can see time apparently jump backwards. For information on a way to avoid this, see **getimeofday** subroutine. A time correction from an earlier call to the **adjtime** subroutine may not be finished when the **adjtime** subroutine

is called again. If the *Olddelta* parameter is nonzero, then the structure pointed to will contain, upon return, the number of microseconds still to be corrected from the earlier call.

This call may be used by time servers that synchronize the clocks of computers in a local area network. Such time servers would slow down the clocks of some machines and speed up the clocks of others to bring them to the average network time.

The **adjtime** subroutine is restricted to the users with root user authority.

Parameters

Item	Description
Delta	Specifies the amount of time to be altered.
Olddelta	Contains the number of microseconds still to be corrected from an earlier call.

Return Values

A return value of 0 indicates that the **adjtime** subroutine succeeded. A return value of -1 indicates than an error occurred, and **errno** is set to indicate the error.

Error Codes

The **adjtime** subroutine fails if the following are true:

Item	Description
EFAULT	An argument address referenced invalid memory.
EPERM	The process's effective user ID does not have root user authority.

agg_proc_stat, agg_lpar_stat, agg_arm_stat, or free_agg_list Subroutine

Purpose

Aggregate advanced accounting data.

Library

The `libaacct.a` library.

Syntax

```
#define <sys/aacct.h>
int agg_arm_stat(tran_list, arm_list);
struct aacct_tran_rec *tran_list
struct agg_arm_stat **arm_list
int agg_proc_stat(sortcrit1, sortcrit2, sortcrit3, sortcrit4, tran_list, proc_list);
int sortcrit1, sortcrit2, sortcrit3, sortcrit4
struct aacct_tran_rec *tran_list
struct agg_proc_stat **proc_list
int agg_lpar_stat(l_type, *tran_list, l_list);
int l_type
struct aacct_tran_rec *tran_list
union agg_lpar_rec *l_list
void free_agg_list(list);
void *list
```

Description

The agg_proc_stat, agg_lpar_stat, and agg_arm_stat subroutines return a linked list of aggregated transaction records for process, LPAR, and ARM, respectively.

The agg_proc_stat subroutine performs the process record aggregation based on the criterion values passed as input parameters. The aggregated process transaction records are sorted based on the sorting criteria values *sortcrit1*, *sortcrit2*, *sortcrit3*, and *sortcrit4*. These four can be one of the following values defined in the sys/aacct.h file:

- CRIT_UID
- CRIT_GID
- CRIT_PROJ
- CRIT_CMD
- CRIT_NONE

The order of their usage determines the sorting order applied to the retrieved aggregated list of process transaction records. For example, the sort criteria values of PROJ_GID, PROJ_PROJ, PROJ_UID, PROJ_NONE first sorts the aggregated list on group IDs, which are further sorted based on project IDs, followed by another level of sorting based on user IDs.

Some of the process transaction records (of type TRID_agg_proc) cannot be aggregated based on group IDs and command names. For such records, agg_proc_stat returns an asterisk (*) character as the command name and a value of -2 as the group ID. This indicates to the caller that these records cannot be aggregated.

If the aggregation is not necessary on a specific criteria, agg_proc_stat returns a value of -1 in the respective field. For example, if the aggregation is not necessary on the group ID (CRIT_GID), the retrieved list of aggregation records has a value of -1 filled in the group ID fields.

The agg_lpar_stat retrieves an aggregated list of LPAR transaction records. Because there are several types of LPAR transaction records, the caller must specify the type of LPAR transaction record that is to be aggregated. The transaction record type can be one of the following values, defined in the sys/aacct.h file:

- AGG_CPUMEM
- AGG_FILESYS
- AGG_NETIF
- AGG_DISK
- AGG_VTARGET
- AGG_VCLIENT

The agg_lpar_stat subroutine uses a union argument of type struct agg_lpar_rec. For this argument, the caller must provide the address of the linked list to which the aggregated records should be returned.

The agg_arm_list retrieves an aggregated list of ARM transaction records from the list of transaction records provided as input. The aggregated transaction records are returned to the caller through the structure pointer of type struct agg_arm_stat.

The free_agg_list subroutine frees the memory allocated to the aggregated records returned by the agg_proc_stat, agg_lpar_stat, or agg_arm_stat subroutine.

Parameters

Item	Description
arm_list	Pointer to the linked list of struct agg_arm_stat nodes to be returned.

Item	Description
l_list	Pointer to `union agg_lpar_rec` address to which the aggregated LPAR records are returned.
l_type	Integer value that represents the type of LPAR resource to be aggregated.
list	Pointer to the aggregated list to be freed.
proc_list	Pointer to the linked list of `struct agg_proc_stat` nodes to be returned.
sortcrit1, *sortcrit2*, *sortcrit3*, *sortcrit4*	Integer values that represent the sorting criteria to be passed to `agg_proc_stat`.
tran_list	Pointer to the input list of transaction records

Security

No restrictions. Any user can call this function.

Return Values

Item	Description
0	The call to the subroutine was successful.
-1	The call to the subroutine failed.

Error Codes

Item	Description
EINVAL	The passed pointer is NULL.
ENOMEM	Insufficient memory.
EPERM	Permission denied. Unable to read the data file.

aio_cancel or aio_cancel64 Subroutine

The **aio_cancel** or **aio_cancel64** subroutine includes information for the POSIX AIO **aio_cancel** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_cancel** subroutine.

POSIX AIO aio_cancel Subroutine

Purpose

Cancels one or more outstanding asynchronous I/O requests.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <aio.h>

int aio_cancel (fildes, aiocbp)
int fildes;
struct aiocb *aiocbp;
```

Description

The **aio_cancel** subroutine cancels one or more asynchronous I/O requests currently outstanding against the *fildes* parameter. The *aiocbp* parameter points to the asynchronous I/O control block for a particular request to be canceled. If *aiocbp* is NULL, all outstanding cancelable asynchronous I/O requests against *fildes* are canceled.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. If there are requests that cannot be canceled, the normal asynchronous completion process takes place for those requests when they are completed.

For requested operations that are successfully canceled, the associated error status is set to **ECANCELED**, and a -1 is returned. For requested operations that are not successfully canceled, the *aiocbp* parameter is not modified by the **aio_cancel** subroutine.

If *aiocbp* is not NULL, and if *fildes* does not have the same value as the file descriptor with which the asynchronous operation was initiated, unspecified results occur.

The implementation of the subroutine defines which operations are cancelable.

Parameters

Item	Description
fildes	Identifies the object to which the outstanding asynchronous I/O requests were originally queued.
aiocbp	Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
int               aio_fildes
off_t             aio_offset
char             *aio_buf
size_t            aio_nbytes
int               aio_reqprio
struct sigevent   aio_sigevent
int               aio_lio_opcode
```

Execution Environment

The **aio_cancel** and **aio_cancel64** subroutines can be called from the process environment only.

Return Values

The **aio_cancel** subroutine returns AIO_CANCELED to the calling process if the requested operation(s) were canceled. AIO_NOTCANCELED is returned if at least one of the requested operations cannot be canceled because it is in progress. In this case, the state of the other operations, referenced in the call to **aio_cancel** is not indicated by the return value of **aio_cancel**. The application may determine the state of affairs for these operations by using the **aio_error** subroutine. AIO_ALLDONE is returned if all of the operations are completed. Otherwise, the subroutine returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

Item	Description
EBADF	The *fildes* parameter is not a valid file descriptor.

Legacy AIO aio_cancel Subroutine

Purpose: Cancels one or more outstanding asynchronous I/O requests.

Library (Legacy AIO aio_cancel Subroutine)

Standard C Library (**libc.a**)

Syntax (Legacy AIO aio_cancel Subroutine)

```
#include <aio.h>
```

aio_cancel (*FileDescriptor*, *aiocbp*)
int *FileDescriptor*;
struct aiocb ***aiocbp**;

aio_cancel64 (*FileDescriptor*, *aiocbp*)
int *FileDescriptor*;
struct aiocb64 ***aiocbp**;

Description (Legacy AIO aio_cancel Subroutine)

The **aio_cancel** subroutine attempts to cancel one or more outstanding asynchronous I/O requests issued on the file associated with the *FileDescriptor* parameter. If the pointer to the **aio control block (aiocb)** structure (the *aiocbp* parameter) is not null, then an attempt is made to cancel the I/O request associated with this **aiocb**. The *aiocbp* parameter used by the thread calling **aix_cancel** must have had its request initiated by this same thread. Otherwise, a -1 is returned and **errno** is set to EINVAL. However, if the *aiocbp* parameter is null, then an attempt is made to cancel all outstanding asynchronous I/O requests associated with the *FileDescriptor* parameter without regard to the initiating thread.

The **aio_cancel64** subroutine is similar to the **aio_cancel** subroutine except that it attempts to cancel outstanding large file enabled asynchronous I/O requests. Large file enabled asynchronous I/O requests make use of the **aiocb64** structure instead of the aiocb structure. The **aiocb64** structure allows asynchronous I/O requests to specify offsets in excess of OFF_MAX (2 gigbytes minus 1).

In the large file enabled programming environment, **aio_cancel** is redefined to be **aio_cancel64**.

When an I/O request is canceled, the **aio_error** subroutine called with the handle to the corresponding **aiocb** structure returns **ECANCELED**.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compilation using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:

```
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Parameters (Legacy AIO aio_cancel Subroutine)

Item	Description
FileDescriptor	Identifies the object to which the outstanding asynchronous I/O requests were originally queued.
aiocbp	Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
struct aiocb
{
       int                aio_whence;
       off_t              aio_offset;
       char               *aio_buf;
       ssize_t            aio_return;
       int                aio_errno;
       size_t             aio_nbytes;
       union {
              int         reqprio;
              struct {
                     int   version:8;
                     int   priority:8;
                     int   cache_hint:16;
              } ext;
       } aio_u1;
       int                aio_flag;
       int                aio_iocpfd;
       aio_handle_t       aio_handle;
}

#define aio_reqprio        aio_u1.reqprio
#define aio_version        aio_u1.ext.version
#define aio_priority       aio_u1.ext.priority
#define aio_cache_hint     aio_u1.ext.cache_hint
```

Execution Environment (Legacy AIO aio_cancel Subroutine)

The **aio_cancel** and **aio_cancel64** subroutines can be called from the process environment only.

Return Values (Legacy AIO aio_cancel Subroutine)

Item	Description
AIO_CANCELED	Indicates that all of the asynchronous I/O requests were canceled successfully. The **aio_error** subroutine call with the handle to the **aiocb** structure of the request will return **ECANCELED**.
AIO_NOTCANCELED	Indicates that the **aio_cancel** subroutine did not cancel one or more outstanding I/O requests. This may happen if an I/O request is already in progress. The corresponding error status of the I/O request is not modified.
AIO_ALLDONE	Indicates that none of the I/O requests is in the queue or in progress.
-1	Indicates that the subroutine was not successful. Sets the **errno** global variable to identify the error.

A return code can be set to the following **errno** value:

Item	Description
EBADF	Indicates that the *FileDescriptor* parameter is not valid.

aio_error or aio_error64 Subroutine

The **aio_error** or **aio_error64** subroutine includes information for the POSIX AIO **aio_error** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_error** subroutine.

POSIX AIO aio_error Subroutine

Purpose

Retrieves error status for an asynchronous I/O operation.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <aio.h>

int aio_error (aiocbp)
const struct aiocb *aiocbp;
```

Description

The **aio_error** subroutine returns the error status associated with the **aiocb** structure. This structure is referenced by the *aiocbp* parameter. The error status for an asynchronous I/O operation is the synchronous I/O **errno** value that would be set by the corresponding **read**, **write**, or **fsync** subroutine. If the subroutine has not yet completed, the error status is equal to **EINPROGRESS**.

Parameters

Item	Description
aiocbp	Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
int               aio_fildes
off_t             aio_offset
char             *aio_buf
size_t            aio_nbytes
int               aio_reqprio
struct sigevent   aio_sigevent
int               aio_lio_opcode
```

Execution Environment

The **aio_error** and **aio_error64** subroutines can be called from the process environment only.

Return Values

If the asynchronous I/O operation has completed successfully, the **aio_error** subroutine returns a 0. If unsuccessful, the error status (as described for the **read**, **write**, and **fsync** subroutines) is returned. If the asynchronous I/O operation has not yet completed, **EINPROGRESS** is returned.

Error Codes

Item	Description
EINVAL	The *aiocbp* parameter does not refer to an asynchronous operation whose return status has not yet been retrieved.

Legacy AIO aio_error Subroutine

Purpose: Retrieves the error status of an asynchronous I/O request.

Library (Legacy AIO aio_error Subroutine)

Standard C Library (**libc.a**)

Syntax (Legacy AIO aio_error Subroutine)

```
#include <aio.h>
```

int
aio_error(*handle***)**
aio_handle_t *handle***;**

int aio_error64(*handle***)**
aio_handle_t *handle***;**

Description (Legacy AIO aio_error Subroutine)

The **aio_error** subroutine retrieves the error status of the asynchronous request associated with the *handle* parameter. The error status is the **errno** value that would be set by the corresponding I/O operation. The error status is **EINPROG** if the I/O operation is still in progress.

The **aio_error64** subroutine is similar to the **aio_error** subroutine except that it retrieves the error status associated with an **aiocb64** control block.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:

```
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Parameters (Legacy AIO aio_error Subroutine)

Item	Description
handle	The handle field of an **aio control block** (**aiocb** or **aiocb64**) structure set by a previous call of the **aio_read**, **aio_read64**, **aio_write**, **aio_write64**, **lio_listio**, **aio_listio64** subroutine. If a random memory location is passed in, random results are returned.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
struct aiocb
{
       int                aio_whence;
       off_t              aio_offset;
       char               *aio_buf;
       ssize_t            aio_return;
       int                aio_errno;
       size_t             aio_nbytes;
       union {
              int         reqprio;
              struct {
                     int   version:8;
                     int   priority:8;
                     int   cache_hint:16;
              } ext;
       } aio_u1;
       int                aio_flag;
       int                aio_iocpfd;
       aio_handle_t       aio_handle;
}

#define aio_reqprio       aio_u1.reqprio
#define aio_version       aio_u1.ext.version
```

```
#define aio_priority       aio_u1.ext.priority
#define aio_cache_hint     aio_u1.ext.cache_hint
```

Execution Environment (Legacy AIO aio_error Subroutine)

The **aio_error** and **aio_error64** subroutines can be called from the process environment only.

Return Values (Legacy AIO aio_error Subroutine)

Item	Description
0	Indicates that the operation completed successfully.
ECANCELED	Indicates that the I/O request was canceled due to an **aio_cancel** subroutine call.
EINPROG	Indicates that the I/O request has not completed. An **errno** value described in the **aio_read** , **aio_write**, and **lio_listio** subroutines: Indicates that the operation was not queued successfully. For example, if the **aio_read** subroutine is called with an unusable file descriptor, it (**aio_read**) returns a value of -1 and sets the **errno** global variable to **EBADF**. A subsequent call of the **aio_error** subroutine with the handle of the unsuccessful **aio control block** (**aiocb**) structure returns **EBADF**. An **errno** value of the corresponding I/O operation: Indicates that the operation was initiated successfully, but the actual I/O operation was unsuccessful. For example, calling the **aio_write** subroutine on a file located in a full file system returns a value of 0, which indicates the request was queued successfully. However, when the I/O operation is complete (that is, when the **aio_error** subroutine no longer returns **EINPROG**), the **aio_error** subroutine returns **ENOSPC**. This indicates that the I/O was unsuccessful.

aio_fsync Subroutine

Purpose

Synchronizes asynchronous files.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <aio.h>

int aio_fsync (op, aiocbp)
int op;
struct aiocb *aiocbp;
```

Description

The **aio_fsync** subroutine asynchronously forces all I/O operations to the synchronized I/O completion state. The function call returns when the synchronization request has been initiated or queued to the file or device (even when the data cannot be synchronized immediately).

If the *op* parameter is set to O_DSYNC, all currently queued I/O operations are completed as if by a call to the **fdatasync** subroutine. If the *op* parameter is set to O_SYNC, all currently queued I/O operations are completed as if by a call to the **fsync** subroutine. If the **aio_fsync** subroutine fails, or if the operation queued by **aio_fsync** fails, outstanding I/O operations are not guaranteed to be completed.

If **aio_fsync** succeeds, it is only the I/O that was queued at the time of the call to **aio_fsync** that is guaranteed to be forced to the relevant completion state. The completion of subsequent I/O on the file descriptor is not guaranteed to be completed in a synchronized fashion.

The *aiocbp* parameter refers to an asynchronous I/O control block. The *aiocbp* value can be used as an argument to the **aio_error** and **aio_return** subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. When the request is queued, the error status for the operation is **EINPROGRESS**. When all data has been successfully transferred, the error status is reset to reflect the success or failure of the operation. If the operation does not complete successfully, the error status for the operation is set to indicate the error. The *aio_sigevent* member determines the asynchronous notification to occur when all operations have achieved synchronized I/O completion. All other members of the structure referenced by the *aiocbp* parameter are ignored. If the control block referenced by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

If the **aio_fsync** subroutine fails or *aiocbp* indicates an error condition, data is not guaranteed to have been successfully transferred.

Parameters

Item	Description
op	Determines the way all currently queued I/O operations are completed.
aiocbp	Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
int               aio_fildes
off_t             aio_offset
char             *aio_buf
size_t            aio_nbytes
int               aio_reqprio
struct sigevent   aio_sigevent
int               aio_lio_opcode
```

Execution Environment

The **aio_error** and **aio_error64** subroutines can be called from the process environment only.

Return Values

The **aio_fsync** subroutine returns a 0 to the calling process if the I/O operation is successfully queued. Otherwise, it returns a -1, and sets the **errno** global variable to indicate the error.

Error Codes

Item	Description
EAGAIN	The requested asynchronous operation was not queued due to temporary resource limitations.
EBADF	The *aio_fildes* member of the **aiocb** structure referenced by the *aiocbp* parameter is not a valid file descriptor open for writing.

In the event that any of the queued I/O operations fail, the **aio_fsync** subroutine returns the error condition defined for the **read** and **write** subroutines. The error is returned in the error status for the asynchronous **fsync** subroutine, which can be retrieved using the **aio_error** subroutine.

aio_nwait Subroutine

Purpose

Suspends the calling process until a certain number of asynchronous I/O requests are completed.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <aio.h>

int aio_nwait (cnt, nwait, list)
int cnt;
int nwait;
struct aiocb **list;
```

Description

Although the **aio_nwait** subroutine is included with POSIX AIO, it is not part of the standard definitions for POSIX AIO.

The **aio_nwait** subroutine suspends the calling process until a certain number (*nwait*) of asynchronous I/O requests are completed. These requests are initiated at an earlier time by the **lio_listio** subroutine, which uses the LIO_NOWAIT_AIOWAIT *cmd* parameter. The **aio_nwait** subroutine fills in the **aiocb** pointers to the completed requests in *list* and returns the number of valid entries in *list*. The *cnt* parameter is the maximum number of **aiocb** pointers that *list* can hold (*cnt* >= *nwait*). The subroutine also returns when less than *nwait* number of requests are done if there are no more pending aio requests.

Note: If the **lio_listio64** subroutine is used, the **aiocb** structure changes to **aiocb64**.

Note: The aio control block's **errno** field continues to have the value EINPROG until after the **aio_nwait** subroutine is completed. The **aio_nwait** subroutine updates this field when the **lio_listio** subroutine has run with the LIO_NOWAIT_AIOWAIT *cmd* parameter. No utility, such as **aio_error**, can be used to look at this value until after the **aio_nwait** subroutine is completed.

The **aio_suspend** subroutine returns after any one of the specified requests gets done. The **aio_nwait** subroutine returns after a certain number (*nwait* or more) of requests are completed.

There are certain limitations associated with the **aio_nwait** subroutine, and a comparison between it and the **aio_suspend** subroutine is necessary. The following table is a comparison of the two subroutines:

aio_suspend:	**aio_nwait:**
Requires users to build a list of control blocks, each associated with an I/O operation they want to wait for.	Requires the user to provide an array to put **aiocb** address information into. No specific aio control blocks need to be known.
Returns when any one of the specified control blocks indicates that the I/O associated with that control block completed.	Returns when *nwait* amount of requests are done or no other requests are to be processed.
The aio control blocks may be updated before the subroutine is called. Other polling methods (such as the **aio_error** subroutine) can also be used to view the aio control blocks.	Updates the aio control blocks itself when it is called. Other polling methods can't be used until after the **aio_nwait** subroutine is called enough times to cover all of the aio requests specified with the **lio_listio** subroutine.

aio_suspend:

aio_nwait:

Is only used in accordance with the **LIO_NOWAIT_AIOWAIT** command, which is one of the commands associated with the **lio_listio** subroutine. If the **lio_listio** subroutine is not first used with the **LIO_NOWAIT_AIOWAIT** command, **aio_nwait** can not be called. The **aio_nwait** subroutine only affects those requests called by one or more **lio_listio** calls for a specified process.

Parameters

Item	Description
cnt	Specifies the number of entries in the list array. This number must be greater than 0 and less than 64 000.
nwait	Specifies the minimal number of requests to wait on. This number must be greater than 0 and less than or equal to the value specified by the *cnt* parameter.
list	An array of pointers to aio control structures defined in the **aio.h** file.

Return Values

The return value is the total number of requests the **aio_nwait** subroutine has waited on to complete. It can not be more than *cnt*. Although *nwait* is the desired amount of requests to find, the actual amount returned could be less than, equal to, or greater than *nwait*. The return value indicates how much of the list array to access.

The return value may be greater than the *nwait* value if the **lio_listio** subroutine initiated more than *nwait* requests and the *cnt* variable is larger than *nwait*. The *nwait* parameter represents a minimal value desired for the return value, and *cnt* is the maximum value possible for the return.

The return value may be less than the *nwait* value if some of the requests initiated by the **lio_listio** subroutine occur at a time of high activity, and there is a lack of resources available for the number of requests. **EAGAIN** (error try again later) may be returned in some request's aio control blocks, but these requests will not be seen by the **aio_nwait** subroutine. In this situation **aiocb** addresses not found on the list have to be accessed by using the **aio_error** subroutine after the **aio_nwait** subroutine is called. You may need to increase the aio parameters *max servers* or *max requests* if this occurs. Increasing the parameters will ensure that the system is well tuned, and an **EAGAIN** error is less likely to occur.

In the event of an error, the **aio_nwait** subroutine returns a value of -1 and sets the **errno** global variable to identify the error. Return codes can be set to the following **errno** values:

Item	Description
EBUSY	An **aio_nwait** call is in process.
EINVAL	The application has retrieved all of the **aiocb** pointers, but the user buffer does not have enough space for them.
EINVAL	There are no outstanding async I/O calls.
EINVAL	Specifies *cnt* or *nwait* values that are not valid.

aio_nwait_timeout Subroutine

Purpose

Extends the capabilities of the **aio_nwait** subroutine by specifying timeout values.

Library

Standard C library (**libc.a**).

Syntax

```
int aio_nwait_timeout (cnt, nwait, list, timeout)
int cnt;
int nwait;
struct aiocbp **list;
int timeout;
```

Description

The **aio_nwait_timeout** subroutine waits for a certain number of asynchronous I/O operations to complete as specified by the *nwait* parameter, or until the call has blocked for a certain duration specified by the *timeout* parameter.

Parameters

Item	Description
cnt	Indicates the maximum number of pointers to the **aiocbp** structure that can be copied into the **list** array.
list	An array of pointers to aio control structures defined in the **aio.h** file.
nwait	Specifies the number of asynchronous I/O operations that must complete before the **aio_nwait_timout** subroutine returns.
timeout	Specified in units of milliseconds. A *timeout* value of `-1` indicates that the subroutine behaves like the **aio_nwait** subroutine, blocking until all of the requested I/O operations complete or until there are no more asynchronous I/O requests pending from the process. A *timeout* value of `0` indicates that the subroutine returns immediately with the current completed number of asynchronous I/O requests. All other positive *timeout* values indicate that the subroutine must block until either the *timeout* value is reached or the requested number of asynchronous I/O operations complete.

Return Values

The return value is the total number of requests the **aio_nwait** subroutine has waited on to complete. It can not be more than *cnt*. Although *nwait* is the desired amount of requests to find, the actual amount returned could be less than, equal to, or greater than *nwait*. The return value indicates how much of the list array to access.

The return value may be greater than the *nwait* value if the **lio_listio** subroutine initiated more than *nwait* requests and the *cnt* variable is larger than *nwait*. The *nwait* parameter represents a minimal value desired for the return value, and *cnt* is the maximum value possible for the return.

The return value may be less than the *nwait* value if some of the requests initiated by the **lio_listio** subroutine occur at a time of high activity, and there is a lack of resources available for the number of requests. The **EAGAIN** return code (error try again later) might be returned in some request's aio control blocks, but these requests will not be seen by the **aio_nwait** subroutine. In this situation, the**aiocb** structure addresses that are not found on the list must be accessed using the **aio_error** subroutine after the **aio_nwait** subroutine is called. You might need to increase the aio parameters max servers or max requests if this occurs. Increasing the parameters will ensure that the system is well tuned, and an **EAGAIN** error is less likely to occur. The return value might be less than the *nwait* value due to the setting of the new timeout parameter in the following cases:

- *timeout* > 0 and a timeout has occurred before nwait requests are done
- *timeout* = 0 and the current requests completed at the time of the **aio_nwait_timeout** call are less then *nwait* parameter

In the event of an error, the **aio_nwait** subroutine returns a value of -1 and sets the **errno** global variable to identify the error. Return codes can be set to the following **errno** values:

Item	Description
EBUSY	An **aio_nwait** call is in process.
EINVAL	The application has retrieved all of the **aiocb** pointers, but the user buffer does not have enough space for them.
EINVAL	There are no outstanding async I/O calls.

aio_read or aio_read64 Subroutine

The **aio_read** or **aio_read64** subroutine includes information for the POSIX AIO **aio_read** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_read** subroutine.

POSIX AIO aio_read Subroutine

Purpose

Asynchronously reads a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <aio.h>

int aio_read (aiocbp)
struct aiocb *aiocbp;
```

Description

The **aio_read** subroutine reads *aio_nbytes* from the file associated with *aio_fildes* into the buffer pointed to by *aio_buf*. The subroutine returns when the read request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

The *aiocbp* value may be used as an argument to the **aio_error** and **aio_return** subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding. If an error condition is encountered during queuing, the function call returns without having initiated or queued the request. The requested operation takes place at the absolute position in the file as given by *aio_offset* , as if the **lseek** subroutine were called immediately prior to the operation with an offset equal to *aio_offset* and a whence equal to SEEK_SET. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aio_lio_opcode* field is ignored by the **aio_read** subroutine.

If prioritized I/O is supported for this file, the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*.

The *aiocbp* parameter points to an **aiocb** structure. If the buffer pointed to by *aio_buf* or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

If synchronized I/O is enabled on the file associated with *aio_fildes*, the behavior of this subroutine is according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding, the result of that action is undefined.

For regular files, no data transfer occurs past the offset maximum established in the open file description.

If you use the **aio_read** or **aio_read64** subroutine with a file descriptor obtained from a call to the **shm_open** subroutine, it will fail with **EINVAL**.

Parameters

Item	Description
aiocbp	Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
int               aio_fildes
off_t             aio_offset
char             *aio_buf
size_t            aio_nbytes
int               aio_reqprio
struct sigevent   aio_sigevent
int               aio_lio_opcode
```

Execution Environment

The **aio_read** and **aio_read64** subroutines can be called from the process environment only.

Return Values

The **aio_read** subroutine returns 0 to the calling process if the I/O operation is successfully queued. Otherwise, it returns a -1 and sets the **errno** global variable to indicate the error.

Error Codes

Item	Description
EAGAIN	The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to the **aio_read** subroutine, or asynchronously. If any of the conditions below are detected synchronously, the **aio_read** subroutine returns -1 and sets the **errno** global variable to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

Item	Description
EBADF	The *aio_fildes* parameter is not a valid file descriptor open for reading.
EINVAL	The file offset value implied by *aio_offset* is invalid, *aio_reqprio* is an invalid value, or *aio_nbytes* is an invalid value. The **aio_read** or **aio_read64** subroutine was used with a file descriptor obtained from a call to the **shm_open** subroutine.

If the **aio_read** subroutine successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is one of the values

normally returned by the **read** subroutine. In addition, the error status of the asynchronous operation is set to one of the error statuses normally set by the **read** subroutine, or one of the following values:

Item	Description
EBADF	The *aio_fildes* argument is not a valid file descriptor open for reading.
ECANCELED	The requested I/O was canceled before the I/O completed due to an explicit **aio_cancel** request.
EINVAL	The file offset value implied by *aio_offset* is invalid.

The following condition may be detected synchronously or asynchronously:

Item	Description
EOVERFLOW	The file is a regular file, *aio_nbytes* is greater than 0, and the starting offset in *aio_offset* is before the end-of-file and is at or beyond the offset maximum in the open file description associated with *aio_fildes*.

Legacy AIO aio_read Subroutine

Purpose: Reads asynchronously from a file.

Library (Legacy AIO aio_read Subroutine)

Standard C Library (**libc.a**)

Syntax (Legacy AIO aio_read Subroutine)

```
#include <aio.h>

int aio_read( FileDescriptor,  aiocbp)
int FileDescriptor;
struct aiocb *aiocbp;

int aio_read64( FileDescriptor,  aiocbp)
int FileDescriptor;
struct aiocb64 *aiocbp;
```

Description (Legacy AIO aio_read Subroutine)

The **aio_read** subroutine reads asynchronously from a file. Specifically, the **aio_read** subroutine reads from the file associated with the *FileDescriptor* parameter into a buffer.

The **aio_read64** subroutine is similar to the **aio_read** subroutine execpt that it takes an **aiocb64** reference parameter. This allows the **aio_read64** subroutine to specify offsets in excess of **OFF_MAX** (2 gigbytes minus 1).

In the large file enabled programming environment, **aio_read** is redefined to be **aio_read64** .

If you use the **aio_read** or **aio_read64** subroutine with a file descriptor obtained from a call to the **shm_open** subroutine, it will fail with **EINVAL**.

The details of the read are provided by information in the **aiocb** structure, which is pointed to by the *aiocbp* parameter. This information includes the following fields:

Item	Description
`aio_buf`	Indicates the buffer to use.
`aio_nbytes`	Indicates the number of bytes to read.

When the read request has been queued, the **aio_read** subroutine updates the file pointer specified by the `aio_whence` and `aio_offset` fields in the **aiocb** structure as if the requested I/O were already completed. It then returns to the calling program. The `aio_whence` and `aio_offset` fields have the same meaning as the *whence* and *offset* parameters in the **lseek** subroutine. The subroutine ignores them for file objects that are not capable of seeking.

If an error occurs during the call, the read request is not queued. To determine the status of a request, use the **aio_error** subroutine.

To have the calling process receive the **SIGIO** signal when the I/O operation completes, set the AIO_SIGNAL bit in the `aio_flag` field in the **aiocb** structure.

Note: The **event** structure in the **aiocb** structure is currently not in use but is included for future compatibility.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:

```
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Since prioritized I/O is not supported at this time, the *aio_reqprio* field of the structure is not presently used.

Parameters (Legacy AIO aio_read Subroutine)

Item	Description
FileDescriptor	Identifies the object to be read as returned from a call to open.
aiocbp	Points to the asynchronous I/O control block structure associated with the I/O operation.

aiocb Structure

The **aiocb** and the **aiocb64** structures are defined in the **aio.h** file and contain the following members:

```
struct aiocb
{
       int                 aio_whence;
       off_t               aio_offset;
       char                *aio_buf;
       ssize_t             aio_return;
       int                 aio_errno;
       size_t              aio_nbytes;
       union {
              int          reqprio;
              struct {
                     int   version:8;
                     int   priority:8;
                     int   cache_hint:16;
              } ext;
       } aio_u1;
       int                 aio_flag;
       int                 aio_iocpfd;
       aio_handle_t        aio_handle;
}

#define aio_reqprio        aio_u1.reqprio
#define aio_version        aio_u1.ext.version
#define aio_priority       aio_u1.ext.priority
#define aio_cache_hint     aio_u1.ext.cache_hint
```

Execution Environment (Legacy AIO aio_read Subroutine)

The **aio_read** and **aio_read64** subroutines can be called from the process environment only.

Return Values (Legacy AIO aio_read Subroutine)

When the read request queues successfully, the **aio_read** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the global variable **errno** to identify the error.

Return codes can be set to the following **errno** values:

Item	Description
EAGAIN	Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may be reached.
EBADF	Indicates that the *FileDescriptor* parameter is not valid.
EFAULT	Indicates that the address specified by the *aiocbp* parameter is not valid.
EINVAL	Indicates that the `aio_whence` field does not have a valid value, or that the resulting pointer is not valid. The **aio_read** or **aio_read64** subroutine was used with a file descriptor obtained from a call to the **shm_open** subroutine.

When using I/O Completion Ports with AIO Requests, return codes can also be set to the following errno values:

Item	Description
EBADF	Indicates that the `aio_iocpfd` field in the aiocb structure is not a valid I/O Completion Port file descriptor.
EINVAL	Indicates that an I/O Completion Port service failed when attempting to start the AIO Request.
EPERM	Indicates that I/O Completion Port services are not available.

Note: Other error codes defined in the **sys/errno.h** file can be returned by the **aio_error** subroutine if an error during the I/O operation is encountered.

aio_return or aio_return64 Subroutine

The **aio_return** or **aio_return64** subroutine includes information for the POSIX AIO **aio_return** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_return** subroutine.

POSIX AIO aio_return Subroutine

Purpose

Retrieves the return status of an asynchronous I/O operation.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <aio.h>

size_t aio_return (aiocbp);
struct aiocb *aiocbp;
```

Description

The **aio_return** subroutine returns the return status associated with the **aiocb** structure. The return status for an asynchronous I/O operation is the value that would be returned by the corresponding **read**, **write**, or **fsync** subroutine call. If the error status for the operation is equal to **EINPROGRESS**, the return status for the operation is undefined. The **aio_return** subroutine can be called once to retrieve the return status of a given asynchronous operation. After that, if the same **aiocb** structure is used in a call to **aio_return** or **aio_error**, an error may be returned. When the **aiocb** structure referred to by *aiocbp* is used to submit another asynchronous operation, the **aio_return** subroutine can be successfully used to retrieve the return status of that operation.

Parameters

Item	Description
aiocbp	Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
int               aio_fildes
off_t             aio_offset
char             *aio_buf
size_t            aio_nbytes
int               aio_reqprio
struct sigevent   aio_sigevent
int               aio_lio_opcode
```

Execution Environment

The **aio_return** and **aio_return64** subroutines can be called from the process environment only.

Return Values

If the asynchronous I/O operation has completed, the return status (as described for the **read**, **write**, and **fsync** subroutines) is returned. If the asynchronous I/O operation has not yet completed, the results of the **aio_return** subroutine are undefined.

Error Codes

Item	Description
EINVAL	The *aiocbp* parameter does not refer to an asynchronous operation whose return status has not yet been retrieved.

Legacy AIO aio_return Subroutine

Purpose: Retrieves the return status of an asynchronous I/O request.

Library (Legacy AIO aio_return Subroutine)

Standard C Library (**libc.a**)

Syntax (Legacy AIO aio_return Subroutine)

```
#include <aio.h>
```

int aio_return(*handle***)**
aio_handle_t *handle***;**

```
int aio_return64( handle)
aio_handle_t handle;
```

Description (Legacy AIO aio_return Subroutine)

The **aio_return** subroutine retrieves the return status of the asynchronous I/O request associated with the **aio_handle_t** handle if the I/O request has completed. The status returned is the same as the status that would be returned by the corresponding **read** or **write** function calls. If the I/O operation has not completed, the returned status is undefined.

The **aio_return64** subroutine is similar to the **aio_return** subroutine except that it retrieves the error status associated with an **aiocb64** control block.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:

```
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Parameters (Legacy AIO aio_return Subroutine)

Item	Description
handle	The handle field of an **aio control block** (**aiocb** or **aiocb64**) structure is set by a previous call of the **aio_read**, **aio_read64**, **aio_write**, **aio_write64**, **lio_listio**, **aio_listio64** subroutine. If a random memory location is passed in, random results are returned.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
struct aiocb
{
       int                 aio_whence;
       off_t               aio_offset;
       char                *aio_buf;
       ssize_t             aio_return;
       int                 aio_errno;
       size_t              aio_nbytes;
       union {
              int          reqprio;
              struct {
                     int   version:8;
                     int   priority:8;
                     int   cache_hint:16;
              } ext;
       } aio_u1;
       int                 aio_flag;
       int                 aio_iocpfd;
       aio_handle_t        aio_handle;
}

#define aio_reqprio        aio_u1.reqprio
#define aio_version        aio_u1.ext.version
#define aio_priority       aio_u1.ext.priority
#define aio_cache_hint     aio_u1.ext.cache_hint
```

Execution Environment (Legacy AIO aio_return Subroutine)

The **aio_return** and **aio_return64** subroutines can be called from the process environment only.

Return Values (Legacy AIO aio_return Subroutine)

The **aio_return** subroutine returns the status of an asynchronous I/O request corresponding to those returned by **read** or **write** functions. If the error status returned by the **aio_error** subroutine call is **EINPROG**, the value returned by the **aio_return** subroutine is undefined.

Examples (Legacy AIO aio_return Subroutine)

An **aio_read** request to read 1000 bytes from a disk device eventually, when the **aio_error** subroutine returns a 0, causes the **aio_return** subroutine to return 1000. An **aio_read** request to read 1000 bytes from a 500 byte file eventually causes the **aio_return** subroutine to return 500. An **aio_write** request to write to a read-only file system results in the **aio_error** subroutine eventually returning **EROFS** and the **aio_return** subroutine returning a value of -1.

aio_suspend or aio_suspend64 Subroutine

The **aio_suspend** subroutine includes information for the POSIX AIO **aio_suspend** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_suspend** subroutine.

POSIX AIO aio_suspend Subroutine

Purpose

Waits for an asynchronous I/O request.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <aio.h>

int aio_suspend (list, nent,
 timeout)
const struct aiocb * const list[];
int nent;
const struct timespec *timeout;
```

Description

The **aio_suspend** subroutine suspends the calling thread until at least one of the asynchronous I/O operations referenced by the *list* parameter has completed, until a signal interrupts the function, or, if timeout is not NULL, until the time interval specified by *timeout* has passed. If any of the **aiocb** structures in the list correspond to completed asynchronous I/O operations (the error status for the operation is not equal to **EINPROGRESS**) at the time of the call, the subroutine returns without suspending the calling thread. The *list* parameter is an array of pointers to asynchronous I/O control blocks. The *nent* parameter indicates the number of elements in the array. Each **aiocb** structure pointed to has been used in initiating an asynchronous I/O request through the **aio_read**, **aio_write**, or **lio_listio** subroutine. This array may contain NULL pointers, which are ignored. If this array contains pointers that refer to **aiocb** structures that have not been used in submitting asynchronous I/O, the effect is undefined.

If the time interval indicated in the **timespec** structure pointed to by *timeout* passes before any of the I/O operations referenced by *list* are completed, the **aio_suspend** subroutine returns with an error. If the Monotonic Clock option is supported, the clock that is used to measure this time interval is the CLOCK_MONOTONIC clock.

Parameters

Item	Description
list	Array of asynchronous I/O operations.
nent	Indicates the number of elements in the *list* array.
timeout	Specifies the time the subroutine has to complete the operation.

Execution Envrionment

The **aio_suspend** and **aio_suspend64** subroutines can be called from the process environment only.

Return Values

If the **aio_suspend** subroutine returns after one or more asynchronous I/O operations have completed, it returns a 0. Otherwise, it returns a -1 and sets the **errno** global variable to indicate the error.

The application can determine which asynchronous I/O completed by scanning the associated error and returning status using the **aio_error** and **aio_return** subroutines, respectively.

Error Codes

Item	Description
EAGAIN	No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by *timeout*.
EINTR	A signal interrupted the **aio_suspend** subroutine. Since each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

Legacy AIO aio_suspend Subroutine

Purpose: Suspends the calling process until one or more asynchronous I/O requests is completed.

Library (Legacy AIO aio_suspend Subroutine)

Standard C Library (**libc.a**)

Syntax (Legacy AIO aio_suspend Subroutine)

```
#include <aio.h>
```

```
aio_suspend( count,  aiocbpa)
int count;
struct aiocb *aiocbpa[ ];
```

```
aio_suspend64( count,  aiocbpa)
int count;
struct aiocb64 *aiocbpa[ ];
```

Description (Legacy AIO aio_suspend Subroutine)

The **aio_suspend** subroutine suspends the calling process until one or more of the *count* parameter asynchronous I/O requests are completed or a signal interrupts the subroutine. Specifically, the **aio_suspend** subroutine handles requests associated with the **aio control block (aiocb)** structures pointed to by the *aiocbpa* parameter.

The **aio_suspend64** subroutine is similar to the **aio_suspend** subroutine except that it takes an array of pointers to **aiocb64** structures. This allows the **aio_suspend64** subroutine to suspend on asynchronous I/O requests submitted by either the **aio_read64**, **aio_write64**, or the **lio_listio64** subroutines.

In the large file enabled programming environment, **aio_suspend** is redefined to be **aio_suspend64**.

The array of **aiocb** pointers may include null pointers, which will be ignored. If one of the I/O requests is already completed at the time of the **aio_suspend** call, the call immediately returns.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:

```
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Parameters (Legacy AIO aio_suspend Subroutine)

Item	Description
count	Specifies the number of entries in the *aiocbpa* array.
aiocbpa	Points to the **aiocb** or **aiocb64** structures associated with the asynchronous I/O operations.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
struct aiocb
{
       int                 aio_whence;
       off_t               aio_offset;
       char                *aio_buf;
       ssize_t             aio_return;
       int                 aio_errno;
       size_t              aio_nbytes;
       union {
              int          reqprio;
              struct {
                     int   version:8;
                     int   priority:8;
                     int   cache_hint:16;
              } ext;
       } aio_u1;
       int                 aio_flag;
       int                 aio_iocpfd;
       aio_handle_t        aio_handle;
}

#define aio_reqprio        aio_u1.reqprio
#define aio_version        aio_u1.ext.version
#define aio_priority       aio_u1.ext.priority
#define aio_cache_hint     aio_u1.ext.cache_hint
```

Execution Envrionment (Legacy AIO aio_suspend Subroutine)

The **aio_suspend** and **aio_suspend64** subroutines can be called from the process environment only.

Return Values (Legacy AIO aio_suspend Subroutine)

If one or more of the I/O requests completes, the **aio_suspend** subroutine returns the index into the aiocbpa array of one of the completed requests. The index of the first element in the *aiocbpa* array is 0. If more than one request has completed, the return value can be the index of any of the completed requests.

In the event of an error, the **aio_suspend** subroutine returns a value of -1 and sets the **errno** global variable to identify the error. Return codes can be set to the following **errno** values:

Item	Description
EINTR	Indicates that a signal or event interrupted the **aio_suspend** subroutine call.
EINVAL	Indicates that the `aio_whence` field does not have a valid value or that the resulting pointer is not valid.

aio_write or aio_write64 Subroutine

The **aio_write** subroutine includes information for the POSIX AIO **aio_write** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **aio_write** subroutine.

POSIX AIO aio_write Subroutine

Purpose

Asynchronously writes to a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <aio.h>

int aio_write (aiocbp)
struct aiocb *aiocbp;
```

Description

The **aio_write** subroutine writes *aio_nbytes* to the file associated with *aio_fildes* from the buffer pointed to by *aio_buf*. The subroutine returns when the write request has been initiated or queued to the file or device.

The *aiocbp* parameter may be used as an argument to the **aio_error** and **aio_return** subroutines in order to determine the error status and return status, respectively, of the asynchronous operation while it is proceeding.

The *aiocbp* parameter points to an **aiocb** structure. If the buffer pointed to by *aio_buf* or the control block pointed to by *aiocbp* becomes an illegal address prior to asynchronous I/O completion, the behavior is undefined.

If **O_APPEND** flag is not set for the *aio_fildes* file descriptor, the requested operation takes place at the absolute position in the file as given by *aio_offset*. This is done as if the **lseek** subroutine were called immediately prior to the operation with an offset equal to *aio_offset* and a whence equal to *SEEK_SET*. If **O_APPEND** flag is set for the file descriptor, write operations append data in bytes to the file in the same order as the calls were made, except under circumstances described in the Asynchronous I/O section in the *System Interfaces and XBD Headers* website. After a successful call to enqueue an asynchronous I/O operation, the value of the file offset for the file is unspecified.

The *aio_lio_opcode* field is ignored by the **aio_write** subroutine.

If prioritized I/O is supported for this file, the asynchronous operation is submitted at a priority equal to the scheduling priority of the process minus *aiocbp->aio_reqprio*.

Simultaneous asynchronous operations using the same *aiocbp* produce undefined results.

If synchronized I/O is enabled on the file associated with *aio_fildes*, the behavior of this subroutine is according to the definitions of synchronized I/O data integrity completion, and synchronized I/O file integrity completion.

For any system action that changes the process memory space while an asynchronous I/O is outstanding, the result of that action is undefined.

For regular files, no data transfer occurs past the offset maximum established in the open file description associated with *aio_fildes*.

If you use the **aio_write** or **aio_write64**subroutine with a file descriptor obtained from a call to the **shm_open** subroutine, it will fail with **EINVAL**.

Parameters

Item	Description
aiocbp	Points to the **aiocb** structure associated with the I/O operation.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
int               aio_fildes
off_t             aio_offset
char             *aio_buf
size_t            aio_nbytes
int               aio_reqprio
struct sigevent   aio_sigevent
int               aio_lio_opcode
```

Execution Environment

The **aio_write** and **aio_write64** subroutines can be called from the process environment only.

Return Values

The **aio_write** subroutine returns a 0 to the calling process if the I/O operation is successfully queued. Otherwise, a -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EAGAIN	The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to **aio_write**, or asynchronously. If any of the conditions below are detected synchronously, the **aio_write** subroutine returns a -1 and sets the **errno** global variable to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the corresponding value.

Item	Description
EBADF	The *aio_fildes* parameter is not a valid file descriptor open for writing.
EINVAL	The file offset value implied by *aio_offset* is invalid, *aio_reqprio* is an invalid value, or *aio_nbytes* is an invalid value. The **aio_write** or **aio_write64** subroutine was used with a file descriptor obtained from a call to the **shm_open** subroutine.

If the **aio_write** subroutine successfully queues the I/O operation, the return status of the asynchronous operation is one of the values normally returned by the **write** subroutine call. If the operation is

successfully queued but is subsequently canceled or encounters an error, the error status for the asynchronous operation contains one of the values normally set by the **write** subroutine call, or one of the following:

Item	Description
EBADF	The *aio_fildes* parameter is not a valid file descriptor open for writing.
EINVAL	The file offset value implied by *aio_offset* would be invalid.
ECANCELED	The requested I/O was canceled before the I/O completed due to an **aio_cancel** request.

The following condition may be detected synchronously or asynchronously:

Item	Description
EFBIG	The file is a regular file, *aio_nbytes* is greater than 0, and the starting offset in *aio_offset* is at or beyond the offset maximum in the open file description associated with *aio_fildes*.

Legacy AIO aio_write Subroutine

Purpose: Writes to a file asynchronously.

Library (Legacy AIO aio_write Subroutine)

Standard C Library (**libc.a**)

Syntax (Legacy AIO aio_write Subroutine)

```
#include <aio.h>

int aio_write( FileDescriptor,  aiocbp)
int FileDescriptor;
struct aiocb *aiocbp;

int aio_write64( FileDescriptor,  aiocbp)
int FileDescriptor;
struct aiocb64 *aiocbp;
```

Description (Legacy AIO aio_write Subroutine)

The **aio_write** subroutine writes asynchronously to a file. Specifically, the **aio_write** subroutine writes to the file associated with the *FileDescriptor* parameter from a buffer. To handle this, the subroutine uses information from the **aio control block (aiocb)** structure, which is pointed to by the *aiocbp* parameter. This information includes the following fields:

Item	Description
`aio_buf`	Indicates the buffer to use.
`aio_nbytes`	Indicates the number of bytes to write.

The **aio_write64** subroutine is similar to the **aio_write** subroutine except that it takes an **aiocb64** reference parameter. This allows the **aio_write64** subroutine to specify offsets in excess of OFF_MAX (2 gigbytes minus 1).

In the large file enabled programming environment, **aio_read** is redefined to be **aio_read64**.

If you use the **aio_write** or **aio_write64** subroutine with a file descriptor obtained from a call to the **shm_open** subroutine, it will fail with **EINVAL**.

When the write request has been queued, the **aio_write** subroutine updates the file pointer specified by the `aio_whence` and `aio_offset` fields in the **aiocb** structure as if the requested I/O completed. It then

returns to the calling program. The `aio_whence` and `aio_offset` fields have the same meaning as the *whence* and *offset* parameters in the **lseek** subroutine. The subroutine ignores them for file objects that are not capable of seeking.

If an error occurs during the call, the write request is not initiated or queued. To determine the status of a request, use the **aio_error** subroutine.

To have the calling process receive the **SIGIO** signal when the I/O operation completes, set the AIO_SIGNAL bit in the `aio_flag` field in the **aiocb** structure.

Note: The **event** structure in the **aiocb** structure is currently not in use but is included for future compatibility.

Note: The _AIO_AIX_SOURCE macro used in **aio.h** must be defined when using **aio.h** to compile an aio application with the Legacy AIO function definitions. The default compile using the **aio.h** file is for an application with the POSIX AIO definitions. In the source file enter:

```
#define _AIO_AIX_SOURCE
#include <sys/aio.h>
```

or, on the command line when compiling enter:

```
->xlc ... -D_AIO_AIX_SOURCE ... legacy_aio_program.c
```

Since prioritized I/O is not supported at this time, the *aio_reqprio* field of the structure is not presently used.

Parameters (Legacy AIO aio_write Subroutine)

Item	Description
FileDescriptor	Identifies the object to be written as returned from a call to open.
aiocbp	Points to the asynchronous I/O control block structure associated with the I/O operation.

aiocb Structure

The **aiocb** structure is defined in the **/usr/include/aio.h** file and contains the following members:

```
struct aiocb
{
       int                 aio_whence;
       off_t               aio_offset;
       char                *aio_buf;
       ssize_t             aio_return;
       int                 aio_errno;
       size_t              aio_nbytes;
       union {
              int          reqprio;
              struct {
                     int   version:8;
                     int   priority:8;
                     int   cache_hint:16;
              } ext;
       } aio_u1;
       int                 aio_flag;
       int                 aio_iocpfd;
       aio_handle_t        aio_handle;
}

#define aio_reqprio        aio_u1.reqprio
#define aio_version        aio_u1.ext.version
#define aio_priority       aio_u1.ext.priority
#define aio_cache_hint     aio_u1.ext.cache_hint
```

Execution Environment (Legacy AIO aio_write Subroutine)

The **aio_write** and **aio_write64** subroutines can be called from the process environment only.

Return Values (Legacy AIO aio_write Subroutine)

When the write request queues successfully, the **aio_write** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **errno** global variable to identify the error.

Return codes can be set to the following **errno** values:

Item	Description
EAGAIN	Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may have been reached.
EBADF	Indicates that the *FileDescriptor* parameter is not valid.
EFAULT	Indicates that the address specified by the *aiocbp* parameter is not valid.
EINVAL	Indicates that the `aio_whence` field does not have a valid value or that the resulting pointer is not valid. The **aio_write** or **aio_write64** subroutine was used with a file descriptor obtained from a call to the **shm_open** subroutine.

When using I/O Completion Ports with AIO Requests, return codes can also be set to the following errno values:

Item	Description
EBADF	Indicates that the `aio_iocpfd` field in the aiocb structure is not a valid I/O Completion Port file descriptor.
EINVAL	Indicates that an I/O Completion Port service failed when attempting to start the AIO Request.
EPERM	Indicates that I/O Completion Port services are not available.

Note: Other error codes defined in the **/usr/include/sys/errno.h** file may be returned by the **aio_error** subroutine if an error during the I/O operation is encountered.

alloc, dealloc, print, read_data, read_regs, symbol_addrs, write_data, and write_regs Subroutine

Purpose

Provide access to facilities needed by the pthread debug library and supplied by the debugger or application.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int alloc (user, len, bufp)
pthdb_user_t user;
size_t len;
void **bufp;
```

```
int dealloc (user, buf)
pthdb_user_t user;
void *buf;
```

```
int print (user, str)
pthdb_user_t user;
char *str;
```

```
int read_data (user, buf, addr, size)
pthdb_user_t user;
void *buf;
pthdb_addr_t addr;
int size;
```

```
int read_regs (user, tid, flags, context)
pthdb_user_t user;
tid_t tid;
unsigned long long flags;
struct context64 *context;
```

```
int symbol_addrs (user, symbols[],count)
pthdb_user_t user;
pthdb_symbol_t symbols[];
int count;
```

```
int write_data (user, buf, addr, size)
pthdb_user_t user;
void *buf;
pthdb_addr_t addr;
int size;
```

```
int write_regs (user, tid, flags, context)
pthdb_user_t user;
tid_t tid;
unsigned long long flags;
struct context64 *context;
```

Description

int alloc()
Allocates *len* bytes of memory and returns the address. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int dealloc()
Takes a buffer and frees it. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int print()
Prints the character string to the debugger's stdout. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back is for debugging the library only. If you aren't debugging the pthread debug library, pass a NULL value for this call back.

int read_data()
Reads the requested number of bytes of data at the requested location from an active process or core file and returns the data through a buffer. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is always required.

int read_regs()
Reads the context information of a debuggee kernel thread from an active process or from a core file. The information should be formatted in **context64** form for both a 32-bit and a 64-bit process. If successful, 0 is returned; otherwise, a nonzero number is returned. This function is only required when using the **pthdb_pthread_context** and **pthdb_pthread_setcontext** subroutines.

int symbol_addrs()
Resolves the address of symbols in the debuggee. The pthread debug library calls this subroutine to get the address of known debug symbols. If the symbol has a name of NULL or "", set the address to 0LL instead of doing a lookup or returning an error. If successful, 0 is returned; otherwise, a nonzero number is returned. In introspective mode, when the **PTHDB_FLAG_SUSPEND** flag is set, the application can use the pthread debug library by passing NULL, or it can use one of its own.

int write_data()
Writes the requested number of bytes of data to the requested location. The **libpthdebug.a** library may use this to write data to the active process. If successful, 0 is returned; otherwise, a nonzero number is returned. This call back function is required when the **PTHDB_FLAG_HOLD** flag is set and when using the **pthdb_pthread_setcontext** subroutine.

int write_regs()
Writes requested context information to specified debuggee's kernel thread id. If successful, 0 is returned; otherwise, a nonzero number is returned. This subroutine is only required when using the **pthdb_pthread_setcontext** subroutine.

Note: If the **write_data** and **write_regs** subroutines are NULL, the pthread debug library will not try to write data or regs. If the **pthdb_pthread_set_context** subroutine is called when the **write_data** and **write_regs** subroutines are NULL, **PTHDB_NOTSUP** is returned.

Parameters

Item	Description
user	User handle.
symbols	Array of symbols.
count	Number of symbols.
buf	Buffer.
addr	Address to be read from or wrote to.
size	Size of the buffer.
flags	Session flags, must accept **PTHDB_FLAG_GPRS**, **PTHDB_FLAG_SPRS**, **PTHDB_FLAG_FPRS**, and **PTHDB_FLAG_REGS**.
tid	Thread id.
flags	Flags that control which registers are read or wrote.
context	Context structure.
len	Length of buffer to be allocated or reallocated.
bufp	Pointer to buffer.
str	String to be printed.

Return Values

If successful, these subroutines return 0; otherwise they return a nonzero value.

alloclmb Subroutine

Purpose

Allocates a contiguous block of contiguous real memory for exclusive use by the caller. The block of memory reserved will be the size of a system LMB.

Syntax

```
#include <sys/dr.h>

int alloclmb(long long *laddr, int flags)
```

Description

The **alloclmb()** subroutine reserves an LMB sized block of contiguous real memory for exclusive use by the caller. It returns the partition logical address of that memory in **laddr*.

alloclmb() is only valid in an LPAR environment, and it fails (with **ENOTSUP**) if called in another environment.

Only a privileged user should call **alloclmb()**.

Parameters

Item	Description
laddr	On successful return, contains the logical address of the allocated LMB.
flags	Must be 0.

Execution Environment

This **alloclmb()** interface should only be called from the process environment.

Return Values

Item	Description
0	The LMB is successfully allocated.

Error Codes

Item	Description
ENOTSUP	LMB allocation not supported on this system.
EINVAL	Invalid flags value.
EINVAL	Not in the process environment.
ENOMEM	A free LMB could not be made available.

asinh, asinhf, asinhl, asinhd32, asinhd64, and asinhd128 Subroutines

Purpose

Computes the inverse hyperbolic sine.

Syntax

```
#include <math.h>

float asinhf (x)
float x;

long double asinhl (x)
long double x;

double asinh ( x)
double x;
_Decimal32 asinhd32 (x)
_Decimal32 x;

_Decimal64 asinhd64 (x)
_Decimal64 x;
```

```
_Decimal128 asinhd128 (x)
_Decimal128 x;
```

Description

The **asinhf**, **asinhl**, **asinh**, **asinhd32**, **asinhd64**, and **asinhd128** subroutines compute the inverse hyperbolic sine of the*x* parameter.

An application wishing to check for error situations should set **errno** to zero and call **fetestexcept(FE_ALL_EXCEPT)** before calling these subroutines. Upon return, if the **errno** global variable is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **asinhf**, **asinhl**, **asinh**, **asinhd32**, **asinhd64**, and **asinhd128** subroutines return the inverse hyperbolic sine of the given argument.

If *x* is NaN, a NaN is returned.

If *x* is 0, or ±Inf, *x* is returned.

If *x* is subnormal, a range error may occur and *x* will be returned.

asinf, asinl, asin, asind32, asind64, and asind128 Subroutines

Purpose

Computes the arc sine.

Syntax

```
#include <math.h>

float asinf (x)
float x;

long double asinl (x)
long double x;

double asin (x)
double x;
_Decimal32 asind32 (x)
_Decimal32 x;

_Decimal64 asind64 (x)
_Decimal64 x;

_Decimal128 asind128 (x)
_Decimal128 x;
```

Description

The **asinf**, **asinl**, **asin**, **asind32**, **asind64**, and **asind128** subroutines compute the principal value of the arc sine of the *x* parameter. The value of *x* should be in the range [-1,1].

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept(FE_ALL_EXCEPT)** before calling these subroutines. On return, if **errno** is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **asinf**, **asinl**, **asin**, **asind32**, **asind64**, and **asind128** subroutines return the arc sine of *x*, in the range [-pi /2, pi/2] radians.

For finite values of *x* not in the range [-1,1], a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is 0, *x* is returned.

If *x* is ±Inf, a domain error occurs, and a NaN is returned.

If *x* is subnormal, a range error may occur and *x* is returned.

assert Macro

Purpose

Verifies a program assertion.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <assert.h>

void assert ( Expression)
int  Expression;
```

Description

The **assert** macro puts error messages into a program. If the specified expression is false, the **assert** macro writes the following message to standard error and stops the program:

```
Assertion failed: Expression, file FileName, line LineNumber
```

In the error message, the *FileName* value is the name of the source file and the *LineNumber* value is the source line number of the **assert** statement.

Parameters

Item	Description
Expression	Specifies an expression that can be evaluated as true or false. This expression is evaluated in the same manner as the C language IF statement.

at_quick_exit Subroutine

Purpose

Registers the function that is specified by the *func* parameter during a call to the **quick_exit** subroutine.

Library

Standard C library (**libc.a**)

Syntax

```
#include <stdlib.h>
int at_quick_exit (void * func (void));
```

Description

The **at_quick_exit** subroutine registers the function that is specified by the *func* parameter that is called without any arguments. If the **quick_exit** subroutine is called, it calls the registered functions before the exit.

If a call to the **at_quick_exit** subroutine does not occur before a call to the **quick_exit** subroutine, the function call is successful.

Parameters

Item	Description
func	Specifies the function that gets registered and that is called during the **quick_exit** subroutine call.

Environmental limits

The implementation supports a minimum registration of up to 32 functions.

Return Values

Upon successful completion, the subroutine returns a value of zero, if the registration succeeds.

If unsuccessful, a value of nonzero is returned.

Files

The **stdlib.h** file defines standard macros, data types, and subroutines.

atan2f, atan2l, atan2, atan2d32, atan2d64, and atan2d128 Subroutines

Purpose

Computes the arc tangent.

Syntax

```
#include <math.h>

float atan2f (y, x)
```

```
float y, float x;

long double atan2l (y, x)
long double y, long double x;

double atan2 (y, x)
double y, x;
_Decimal32 atan2d32 (y, x)
_Decimal32 y, x;

_Decimal64 atan2d64 (y, x)
_Decimal64 y, x;

_Decimal128 atan2d128 (y, x)
_Decimal128 y, x;
```

Description

The **atan2f**, **atan2l**, **atan2**, **atan2d32**, **atan2d64** and **atan2d128** subroutines compute the principal value of the arc tangent of *y/x*, using the signs of both parameters to determine the quadrant of the return value.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept(FE_ALL_EXCEPT)** before calling these functions. On return, if **errno** is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

Item	Description
y	Specifies the value to compute.
x	Specifies the value to compute.

Return Values

Upon successful completion, the **atan2f**, **atan2l**, **atan2**, **atan2d32**, **atan2d64**, and **atan2d128** subroutines return the arc tangent of *y/x* in the range [-pi, pi] radians.

If *y* is 0 and *x* is < 0, ±pi is returned.

If *y* is 0 and *x* is > 0, 0 is returned.

If *y* is < 0 and *x* is 0, -pi/2 is returned.

If *y* is > 0 and *x* is 0, pi/2 is returned.

If *x* is 0, a pole error does not occur.

If either *x* or *y* is NaN, a NaN is returned.

If the result underflows, a range error may occur and *y/x* is returned.

If *y* is 0 and *x* is -0, ±x is returned.

If *y* is 0 and *x* is +0, 0 is returned.

For finite values of ±*y* >0, if *x* is -Inf, ±x is returned.

For finite values of ±*y* >0, if *x* is +Inf, 0 is returned.

For finite values of *x*, if *y* is ±Inf, ±x/2 is returned.

If *y* is ±Inf and *x* is -Inf, ±3pi/4 is returned.

If *y* is ±Inf and *x* is +Inf, ±pi/4 is returned.

If both arguments are 0, a domain error does not occur.

atan, atanf, atanl, atand32, atand64, and atand128 Subroutines

Purpose

Computes the arc tangent.

Syntax

```
#include <math.h>

float atanf (x)
float x;

long double atanl (x)
long double x;

double atan (x)
double x;
_Decimal32 atand32 (x)
_Decimal32 x;

_Decimal64 atand64 (x)
_Decimal64 x;

_Decimal128 atand128 (x)
_Decimal128 x;
```

Description

The **atanf**, **atanl**, **atan**, **atand32**, **atand64**, and **atand128** subroutines compute the principal value of the arc tangent of the *x* parameter.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept(FE_ALL_EXCEPT)** before calling these functions. On return, if **errno** is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **atanf**, **atanl**, **atan**, **atand32**, **atand64**, and **atand128** subroutines return the arc tangent of *x* in the range [-pi /2, pi/2] radians.

If *x* is NaN, a NaN is returned.

If *x* is 0, *x* is returned.

If *x* is ±Inf, ±x/2 is returned.

If *x* is subnormal, a range error may occur and *x* is returned.

atanh, atanhf, atanhl, atanhd32, atanhd64, and atanhd128 Subroutines

Purpose

Computes the inverse hyperbolic tangent.

Syntax

```
#include <math.h>

float atanhf (x)
float x;

long double atanhl (x)
long double x;

double atanh (x)
double x;
_Decimal32 atanhd32 (x)
_Decimal32 x;

_Decimal64 atanhd64 (x)
_Decimal64 x;

_Decimal128 atanhd128 (x)
_Decimal128 x;
```

Description

The **atanhf**, **atanhl**, **atanh**, **atanhd32**, **atanhd64**, and **atanhd128** subroutines compute the inverse hyperbolic tangent of the *x* parameter.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept(FE_ALL_EXCEPT)** before calling these functions. On return, if **errno** is nonzero or **fetestexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **atanhf**, **atanhl**, **atanh**, **atanhd32**, **atanhd64**, and **atanhd128** subroutines return the inverse hyperbolic tangent of the given argument.

If *x* is ±1, a pole error occurs, and **atanhf**, **atanhl** , **atanh**, **atanhd32**, **atanhd64**, and **atanhd128** return the value of the macro HUGE_VALF, HUGE_VALL, HUGE_VAL, HUGE_VAL_D32, HUGE_VAL_D64, and HUGE_VAL_D128 respectively, with the same sign as the correct value of the function.

For finite |*x*|>1, a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is 0, *x* is returned.

If *x* is ±Inf, a domain error shall occur, and a NaN is returned.

If *x* is subnormal, a range error may occur and *x* is returned.

Error Codes

The **atanhf**, **atanhl**, **atanh**, **atanhd32**, **atanhd64**, and **atanhd128** subroutines return **NaNQ** and set **errno** to **EDOM** if the absolute value of x is greater than the value of one.

atof atoff Subroutine

Purpose

Converts an ASCII string to a floating-point or double floating-point number.

Libraries

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
double atof (NumberPointer)
const char *NumberPointer;
```

```
float atoff (NumberPointer)
char *NumberPointer;
```

Description

The **atof** subroutine converts a character string, pointed to by the *NumberPointer* parameter, to a double-precision floating-point number. The **atoff** subroutine converts a character string, pointed to by the *NumberPointer* parameter, to a single-precision floating-point number. The first unrecognized character ends the conversion.

Except for behavior on error, the **atof** subroutine is equivalent to the **strtod** subroutine call, with the *EndPointer* parameter set to (**char****) NULL.

Except for behavior on error, the **atoff** subroutine is equivalent to the **strtof** subroutine call, with the *EndPointer* parameter set to (**char****) NULL.

These subroutines recognize a character string when the characters are in one of two formats: numbers or numeric symbols.

- For a string to be recognized as a number, it should contain the following pieces in the following order:
  1. An optional string of white-space characters
  2. An optional sign
  3. A nonempty string of digits optionally containing a radix character
  4. An optional exponent in E-format or e-format followed by an optionally signed integer.
- For a string to be recognized as a numeric symbol, it should contain the following pieces in the following order:
  1. An optional string of white-space characters
  2. An optional sign
  3. One of the strings: **INF**, **infinity**, **NaNQ**, **NaNS**, or **NaN** (case insensitive)

The **atoff** subroutine is not part of the ANSI C Library. These subroutines are at least as accurate as required by the *IEEE Standard for Binary Floating-Point Arithmetic*. The **atof** subroutine accepts at least 17 significant decimal digits. The **atoff** and subroutine accepts at least 9 leading 0's. Leading 0's are not counted as significant digits.

Note: Starting with the IBM® AIX 6 with Technology Level 7 and the IBM AIX 7 with Technology Level 1, the precision of the floating-point conversion routines, printf and scanf family of functions has been increased from 17 digits to 37 digits for double and long double values.

Parameters

Item	Description
NumberPointer	Specifies a character string to convert.
EndPointer	Specifies a pointer to the character that ended the scan or a null value.

Return Values

Upon successful completion, the **atof**, and **atoff** subroutines return the converted value. If no conversion could be performed, a value of 0 is returned and the **errno** global variable is set to indicate the error.

Error Codes

If the conversion cannot be performed, a value of 0 is returned, and the **errno** global variable is set to indicate the error.

If the conversion causes an overflow (that is, the value is outside the range of representable values), **+/- HUGE_VAL** is returned with the sign indicating the direction of the overflow, and the **errno** global variable is set to **ERANGE**.

If the conversion would cause an underflow, a properly signed value of 0 is returned and the **errno** global variable is set to **ERANGE**.

The **atoff** subroutine has only one rounding error. (If the **atof** subroutine is used to create a double-precision floating-point number and then that double-precision number is converted to a floating-point number, two rounding errors could occur.)

atol or atoll Subroutine

Purpose

Converts a string to a long integer.

Syntax

```
#include <stdlib.h>

long long atoll (nptr)
const char *nptr;

long atol (nptr)
const char *nptr;
```

Description

The **atoll** and **atol** subroutines (*str*) are equivalent to `strtoll(nptr, (char **)NULL, 10)` and `strtol(nptr, (char **)NULL, 10)`, respectively. If the value cannot be represented, the behavior is undefined.

Parameters

Item	Description
nptr	Points to the string to be converted into a long integer.

Return Values

The **atoll** and **atol** subroutines return the converted value if the value can be represented.

attrset or wattrset Subroutine

Purpose

Sets the current attributes of a window to the specified attributes.

Libraries

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

attrset( Attributes)
char *Attributes;
wattrset( Window, Attributes)
WINDOW *Window;
char *Attributes;
```

Description

The **attrset** and **wattrset** subroutines set the current attributes of a window to the specified attributes. The **attrset** subroutine sets the current attribute of stdscr. The **wattrset** subroutine sets the current attribute of the specified window.

Parameters

Item	Description
Attributes	Specifies which attributes to set.
Window	Specifies the window in which to set the attributes.

Examples

1. To set the current attribute in the **stdscr** global variable to blink, enter:

   ```
   attrset(A_BLINK);
   ```

2. To set the current attribute in the user-defined window `my_window` to blinking, enter:

   ```
   wattrset(my_window, A_BLINK);
   ```

3. To turn off all attributes in the **stdscr** global variable, enter:

   ```
   attrset(0);
   ```

4. To turn off all attributes in the user-defined window `my_window`, enter:

   ```
   wattrset(my_window, 0);
   ```

attroff, attron, attrset, wattroff, wattron, or wattrset Subroutine

Purpose

Restricted window attribute control functions.

Library

Curses Library (**libcurses.a**)

Syntax

#include <curses.h>

int attroff (int **attrs****);**

int attron (int **attrs****);**

int attrset (int **attrs****);**

int wattroff (WINDOW **win*, **int** **attsr****);**

int wattron (WINDOW **win*, **int** **attrs****);**

int wattrset (WINDOW **win*, **int** **attsr****);**

Description

These subroutines manipulate the window attributes of the current or specified window.

The **attroff** and **wattroff** subroutines turn off *attrs* in the current or specified specified window without affecting any others.

The **attron** and **wattron** subroutines turn on *attrs* in the current or specified specified window without affecting any others.

The **attrset** and **wattrset** subroutines set the background attributes of the current or specified specified window to *attrs.*

It unspecified whether these subroutines can be used to manipulate attributes than A_BLINK, A_BOLD, A_DIM, A_REVERSE, A_STANDOUT and A_UNDERLINE.

Parameters

Item	Description
**attrs*	Specifies which attributes to turn off.
**win*	Specifies the window in which to turn off the specified attributes.

Return Values

These subroutines always return either OK or 1.

Examples

For the **attroff** or **wattroff** subroutines:

1. To turn the off underlining attribute in stdscr, enter:

```
attroff(A_UNDERLINE);
```

2. To turn off the underlining attribute in the user-defined window `my_window`, enter:

```
wattroff(my_window, A_UNDERLINE);
```

For the **attron** or **wattron** subroutines:

1. To turn on the underlining attribute in stdscr, enter:

```
attron(A_UNDERLINE);
```

2. To turn on the underlining attribute in the user-defined window `my_window`, enter:

```
wattron(my_window, A_UNDERLINE);
```

For the **attrset** or **wattrset** subroutines:

1. To set the current attribute in the **stdscr** global variable to blink, enter:

```
attrset(A_BLINK);
```

2. To set the current attribute in the user-defined window `my_window` to blinking, enter:

```
wattrset(my_window, A_BLINK);
```

3. To turn off all attributes in the **stdscr** global variable, enter:

```
attrset(0);
```

4. To turn off all attributes in the user-defined window `my_window`, enter:

```
wattrset(my_window, 0);
```

attron or wattron Subroutine

Purpose

Turns on specified attributes.

Library

Curses Library (**libcurses.a**)

Syntax

#include <curses.h>

```
attron( Attributes)
char *Attributes;
```

```
wattron( Window, Attributes)
WINDOW *Window;
char *Attributes;
```

Description

The **attron** and **wattron** subroutines turn on specified attributes without affecting any others. The **attron** subroutine turns the specified attributes on in stdscr. The **wattron** subroutine turns the specified attributes on in the specified window.

Parameters

Item	Description
Attributes	Specifies which attributes to turn on.
Window	Specifies the window in which to turn on the specified attributes.

Examples

1. To turn on the underlining attribute in stdscr, enter:

```
attron(A_UNDERLINE);
```

2. To turn on the underlining attribute in the user-defined window `my_window`, enter:

```
wattron(my_window, A_UNDERLINE);
```

audit Subroutine

Purpose

Enables and disables system auditing.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/audit.h>
```

```
int audit ( Command,  Argument)
int Command;
int Argument;
```

Description

The **audit** subroutine enables or disables system auditing.

When auditing is enabled, audit records are created for security-relevant events. These records can be collected through the **auditbin** subroutine, or through the **/dev/audit** special file interface.

Parameters

Item	Description
Command	Defined in the **sys/audit.h** file, can be one of the following values: **AUDIT_QUERY** Returns a mask indicating the state of the auditing subsystem. The mask is a logical ORing of the **AUDIT_ON**, **AUDIT_OFF**, **AUDIT_PANIC**, and **AUDIT_FULLPATH** flags. **AUDIT_ON** Enables auditing. If auditing is already enabled, only the failure-mode behavior changes. The *Argument* parameter specifies recovery behavior in the event of failure and may be either 0 or the value **AUDIT_PANIC** or **AUDIT_FULLPATH**. **Note:** If **AUDIT_PANIC** is specified, bin-mode auditing must be enabled before the **audit** subroutine call. **AUDIT_OFF** Disables the auditing system if auditing is enabled. If the auditing system is disabled, the **audit** subroutine does nothing. The *Argument* parameter is ignored. **AUDIT_RESET** Disables the auditing system and resets the auditing system. If auditing is already disabled, only the system configuration is reset. Resetting the audit configuration involves clearing the audit events and audited objects table, and terminating bin auditing and stream auditing. **AUDIT_EVENT_THRESHOLD** Audit event records will be buffered until a total of *Argument* records have been saved, at which time the audit event records will be flushed to disk. An *Argument* value of zero disables this functionality. **AUDIT_BYTE_THRESHOLD** Audit event data will be buffered until a total of *Argument* bytes of data have been saved, at which time the audit event data will be flushed to disk. An *Argument* value of zero disables this functionality.
Argument	Specifies the behavior when a bin write fails (for **AUDIT_ON**) or specifies the size of the audit event buffer (for **AUDIT_EVENT_THRESHOLD** and **AUDIT_BYTE_THRESHOLD**). For **AUDIT_RESET** and **AUDIT_QUERY**, the value of the **Argument** is the WPAR ID. For all other commands, the value of **Argument** is ignored. The valid values are: **AUDIT_PANIC** The operating system halts abruptly if an audit record cannot be written to a bin. **Note:** If **AUDIT_PANIC** is specified, bin-mode auditing must be enabled before the **audit** subroutine call. **AUDIT_FULLPATH** The operating system starts capturing full path name for the FILE_Open, FILE_Read, FILE_Write auditing events. **BufferSize** The number of bytes or audit event records which will be buffered. This parameter is valid only with the command **AUDIT_BYTE_THRESHOLD** and **AUDIT_EVENT_THRESHOLD**. A value of zero will disable either byte (for **AUDIT_BYTE_THRESHOLD**) or event (for **AUDIT_EVENT_THRESHOLD**) buffering.

Return Values

For a *Command* value of **AUDIT_QUERY**, the **audit** subroutine returns, upon successful completion, a mask indicating the state of the auditing subsystem. The mask is a logical ORing of the **AUDIT_ON**, **AUDIT_OFF**, **AUDIT_PANIC**, **AUDIT_NO_PANIC**, and **AUDIT_FULLPATH** flags. For any other *Command* value, the **audit** subroutine returns 0 on successful completion.

If the **audit** subroutine fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **audit** subroutine fails if either of the following is true:

Item	Description
EINVAL	The *Command* parameter is not one of **AUDIT_ON**, **AUDIT_OFF**, **AUDIT_RESET**, or **AUDIT_QUERY**.
EINVAL	The *Command* parameter is **AUDIT_ON** and the *Argument* parameter specifies values other than **AUDIT_PANIC** or **AUDIT_FULLPATH**.
EPERM	The calling process does not have root user authority.

Files

Item	Description
dev/audit	Specifies the audit pseudo-device from which the audit records are read.

auditbin Subroutine

Purpose

Defines files to contain audit records.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/audit.h>

int auditbin (Command, Current, Next, Threshold)
int  Command;
int  Current;
int  Next;
int  Threshold;
```

Description

The **auditbin** subroutine establishes an audit bin file into which the kernel writes audit records. Optionally, this subroutine can be used to establish an overflow bin into which records are written when the current bin reaches the size specified by the *Threshold* parameter.

Parameters

Item	Description
Command	If nonzero, this parameter is a logical ORing of the following values, which are defined in the **sys/audit.h** file: **AUDIT_EXCL** Requests exclusive rights to the audit bin files. If the file specified by the *Current* parameter is not the kernel's current bin file, the **auditbin** subroutine fails immediately with the **errno** variable set to **EBUSY**. **AUDIT_WAIT** The **auditbin** subroutine should not return until: **bin full** The kernel writes the number of bytes specified by the *Threshold* parameter to the file descriptor specified by the *Current* parameter. Upon successful completion, the **auditbin** subroutine returns a 0. The kernel writes subsequent audit records to the file descriptor specified by the *Next* parameter. **bin failure** An attempt to write an audit record to the file specified by the *Current* parameter fails. If this occurs, the **auditbin** subroutine fails with the **errno** variable set to the return code from the **auditwrite** subroutine. **bin contention** Another process has already issued a successful call to the **auditbin** subroutine. If this occurs, the **auditbin** subroutine fails with the **errno** variable set to **EBUSY**. **system shutdown** The auditing system was shut down. If this occurs, the **auditbin** subroutine fails with the **errno** variable set to **EINTR**.
Current	A file descriptor for a file to which the kernel should immediately write audit records.
Next	Specifies the file descriptor that will be used as the current audit bin if the value of the *Threshold* parameter is exceeded or if a write to the current bin fails. If this value is -1, no switch occurs.
Threshold	Specifies the maximum size of the current bin. If 0, the auditing subsystem will not switch bins. If it is nonzero, the kernel begins writing records to the file specified by the *Next* parameter, if writing a record to the file specified by the *Cur* parameter would cause the size of this file to exceed the number of bytes specified by the *Threshold* parameter. If no next bin is defined and **AUDIT_PANIC** was specified when the auditing subsystem was enabled, the system is shut down. If the size of the *Threshold* parameter is too small to contain a bin header and a bin tail, the **auditbin** subroutine fails and the **errno** variable is set to **EINVAL**.

Return Values

If the **auditbin** subroutine is successful, a value of 0 returns.

If the **auditbin** subroutine fails, a value of -1 returns and the **errno** global variable is set to indicate the error. If this occurs, the result of the call does not indicate whether any records were written to the bin.

Error Codes

The **auditbin** subroutine fails if any of the following is true:

Item	Description
EBADF	The *Current* parameter is not a file descriptor for a regular file open for writing, or the *Next* parameter is neither -1 nor a file descriptor for a regular file open for writing.
EBUSY	The *Command* parameter specifies **AUDIT_EXCL** and the kernel is not writing audit records to the file specified by the *Current* parameter.
EBUSY	The *Command* parameter specifies **AUDIT_WAIT** and another process has already registered a bin.
EINTR	The auditing subsystem is shut down.
EINVAL	The *Command* parameter specifies a nonzero value other than **AUDIT_EXCL** or **AUDIT_WAIT**.
EINVAL	The *Threshold* parameter value is less than the size of a bin header and trailer.
EPERM	The caller does not have root user authority.

auditevents Subroutine

Purpose

Gets or sets the status of system event auditing.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/audit.h>
```

```
int auditevents ( Command,  Classes,  NClasses)
int Command;
struct audit_class *Classes;
int NClasses;
```

Description

The **auditevents** subroutine queries or sets the audit class definitions that control event auditing. Each audit class is a set of one or more audit events.

System auditing need not be enabled before calling the **auditevents** subroutine. The **audit** subroutine can be directed with the **AUDIT_RESET** command to clear all event lists.

Parameters

Item	Description
Command	Specifies whether the event lists are to be queried or set. The values, defined in the **sys/audit.h** file, for the *Command* parameter are: **AUDIT_SET** Sets the lists of audited events after first clearing all previous definitions. **AUDIT_GET** Queries the lists of audited events. **AUDIT_LOCK** Queries the lists of audited events. This value also blocks any other process attempting to set or lock the list of audit events. The lock is released when the process holding the lock dies or calls the **auditevents** subroutine with the *Command* parameter set to **AUDIT_SET**.
Classes	Specifies the array of **a_event** structures for the **AUDIT_SET** operation, or after an **AUDIT_GET** or **AUDIT_LOCK** operation. The **audit_class** structure is defined in the **sys/audit.h** file and contains the following members: **ae_name** A pointer to the name of the audit class. **ae_list** A pointer to a list of null-terminated audit event names for this audit class. The list is ended by a null name (a leading null byte or two consecutive null bytes). **Note:** Event and class names are limited to 15 significant characters. **ae_len** The length of the event list in the **ae_list** member. This length includes the terminating null bytes. On an **AUDIT_SET** operation, the caller must set this member to indicate the actual length of the list (in bytes) pointed to by `ae_list`. On an **AUDIT_GET** or **AUDIT_LOCK** operation, the **auditevents** subroutine sets this member to indicate the actual size of the list.
NClasses	Serves a dual purpose. For **AUDIT_SET**, the *NClasses* parameter specifies the number of elements in the events array. For **AUDIT_GET** and **AUDIT_LOCK**, the *NClasses* parameter specifies the size of the buffer pointed to by the *Classes* parameter.

Attention: Only 32 audit classes are supported. One class is implicitly defined by the system to include all audit events (ALL). The administrator of your system should not attempt to define more than 31 audit classes.

Security

The calling process must have root user authority in order to use the **auditevents** subroutine.

Return Codes

If the **auditevents** subroutine completes successfully, the number of audit classes is returned if the *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**. A value of 0 is returned if the *Command* parameter is **AUDIT_SET**. If this call fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditevents** subroutine fails if one or more of the following are true:

Item	Description
EPERM	The calling process does not have root user authority.
EINVAL	The value of *Command* is not **AUDIT_SET**, **AUDIT_GET**, or **AUDIT_LOCK**.
EINVAL	The *Command* parameter is **AUDIT_SET**, and the value of the *NClasses* parameter is greater than or equal to 32.
EINVAL	A class name or event name is longer than 15 significant characters.
ENOSPC	The value of *Command* is **AUDIT_GET** or **AUDIT_LOCK** and the size of the buffer specified by the *NClasses* parameter is not large enough to hold the list of event structures and names. If this occurs, the first word of the buffer is set to the required buffer size.
EFAULT	The *Classes* parameter points outside of the process' address space.
EFAULT	The `ae_list` member of one or more **audit_class** structures passed for an **AUDIT_SET** operation points outside of the process' address space.
EFAULT	The *Command* value is **AUDIT_GET** or **AUDIT_LOCK** and the size of the *Classes* buffer is not large enough to hold an integer.
EBUSY	Another process has already called the **auditevents** subroutine with **AUDIT_LOCK**.
ENOMEM	Memory allocation failed.

auditlog Subroutine

Purpose

Appends an audit record to the audit trail file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/audit.h>

int auditlog ( Event,  Result,  Buffer,  BufferSize)
char *Event;
int Result;
char *Buffer;
int BufferSize;
```

Description

The **auditlog** subroutine generates an audit record. The kernel audit-logging component appends a record for the specified *Event* if system auditing is enabled, process auditing is not suspended, and the *Event* parameter is in one or more of the audit classes for the current process.

The audit logger generates the audit record by adding the *Event* and *Result* parameters to the audit header and including the resulting information in the *Buffer* parameter as the audit tail.

Parameters

Item	Description
Event	The name of the audit event to be generated. This parameter should be the name of an audit event. Audit event names are truncated to 15 characters plus null.
Result	Describes the result of this event. Valid values are defined in the **sys/audit.h** file and include the following: **AUDIT_OK** The event was successful. **AUDIT_FAIL** The event failed. **AUDIT_FAIL_ACCESS** The event failed because of any access control denial. **AUDIT_FAIL_DAC** The event failed because of a discretionary access control denial. **AUDIT_FAIL_PRIV** The event failed because of a privilege control denial. **AUDIT_FAIL_AUTH** The event failed because of an authentication denial. Other nonzero values of the *Result* parameter are converted into the **AUDIT_FAIL** value.
Buffer	Points to a buffer containing the tail of the audit record. The format of the information in this buffer depends on the event name.
BufferSize	Specifies the size of the *Buffer* parameter, including the terminating null.

Return Values

Upon successful completion, the **auditlog** subroutine returns a value of 0. If **auditlog** fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

The **auditlog** subroutine does not return any indication of failure to write the record where this is due to inappropriate tailoring of auditing subsystem configuration files or user-written code. Accidental omissions and typographical errors in the configuration are potential causes of such a failure.

Error Codes

The **auditlog** subroutine fails if any of the following are true:

Item	Description
EFAULT	The *Event* or *Buffer* parameter points outside of the process' address space.
EINVAL	The auditing system is either interrupted or not initialized.
EINVAL	The length of the audit record is greater than 32 kilobytes.
EPERM	The process does not have root user authority.
ENOMEM	Memory allocation failed.

auditobj Subroutine

Purpose

Gets or sets the auditing mode of a system data object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/audit.h>

int auditobj ( Command,  Obj_Events,  ObjSize)
int Command;
struct o_event *Obj_Events;
int ObjSize;
```

Description

The **auditobj** subroutine queries or sets the audit events to be generated by accessing selected objects. For each object in the file system name space, it is possible to specify the event generated for each access mode. Using the **auditobj** subroutine, an administrator can define new audit events in the system that correspond to accesses to specified objects. These events are treated the same as system-defined events.

System auditing need not be enabled to set or query the object audit events. The **audit** subroutine can be directed with the **AUDIT_RESET** command to clear the definitions of object audit events.

Parameters

Item	Description
Command	Specifies whether the object audit event lists are to be read or written. The valid values, defined in the **sys**/**audit.h** file, for the *Command* parameter are: **AUDIT_SET** Sets the list of object audit events, after first clearing all previous definitions. **AUDIT_GET** Queries the list of object audit events. **AUDIT_LOCK** Queries the list of object audit events and also blocks any other process attempting to set or lock the list of audit events. The lock is released when the process holding the lock dies or calls the **auditobj** subroutine with the *Command* parameter set to **AUDIT_SET**.

Item	Description
Obj_Events	Specifies the array of **o_event** structures for the **AUDIT_SET** operation or for after the **AUDIT_GET** or **AUDIT_LOCK** operation. The **o_event** structure is defined in the **sys/audit.h** file and contains the following members: **o_type** Specifies the type of the object, in terms of naming space. Currently, only one object-naming space is supported: **AUDIT_FILE** Denotes the file system naming space. **o_name** Specifies the name of the object. **o_event** Specifies any array of event names to be generated when the object is accessed. Note that event names are currently limited to 16 bytes, including the trailing null. The index of an event name in this array corresponds to an access mode. Valid indexes are defined in the **audit.h** file and include the following: • **AUDIT_READ** • **AUDIT_WRITE** • **AUDIT_EXEC** **Note:** The C++ compiler will generate a warning indicating that **o_event** is defined both as a structure and a field within that structure. Although the **o_event** field can be used within C++, the warning can by bypassed by defining **O_EVENT_RENAME**. This will replace the **o_event** field with **o_event_array**. **o_event** is the default field.
ObjSize	For an **AUDIT_SET** operation, the *ObjSize* parameter specifies the number of object audit event definitions in the array pointed to by the *Obj_Events* parameter. For an **AUDIT_GET** or **AUDIT_LOCK** operation, the *ObjSize* parameter specifies the size of the buffer pointed to by the *Obj_Events* parameter.

Return Values

If the **auditobj** subroutine completes successfully, the number of object audit event definitions is returned if the *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**. A value of 0 is returned if the *Command* parameter is **AUDIT_SET**. If this call fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditobj** subroutine fails if any of the following are true:

Item	Description
EFAULT	The *Obj_Events* parameter points outside the address space of the process.
EFAULT	The *Command* parameter is **AUDIT_SET**, and one or more of the o_name members points outside the address space of the process.
EFAULT	The *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**, and the buffer size of the *Obj_Events* parameter is not large enough to hold the integer.
EINVAL	The value of the *Command* parameter is not **AUDIT_SET**, **AUDIT_GET** or **AUDIT_LOCK**.

Item	Description
EINVAL	The *Command* parameter is **AUDIT_SET**, and the value of one or more of the o_type members is not **AUDIT_FILE**.
EINVAL	An event name was longer than 15 significant characters.
ENOENT	The *Command* parameter is **AUDIT_SET**, and the parent directory of one of the file-system objects does not exist.
ENOSPC	The value of the *Command* parameter is **AUDIT_GET** or **AUDIT_LOCK**, and the size of the buffer as specified by the *ObjSize* parameter is not large enough to hold the list of event structures and names. If this occurs, the first word of the buffer is set to the required buffer size.
ENOMEM	Memory allocation failed.
EBUSY	Another process has called the **auditobj** subroutine with **AUDIT_LOCK**.
EPERM	The caller does not have root user authority.

auditpack Subroutine

Purpose

Compresses and uncompresses audit bins.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>
```

char *auditpack (*Expand*, *Buffer*)
int *Expand*;
char **Buffer*;

Description

The **auditpack** subroutine can be used to compress or uncompress bins of audit records.

Parameters

Item	Description
Expand	Specifies the operation. Valid values, as defined in the **sys/audit.h** header file, are one of the following: **AUDIT_PACK** Performs standard compression on the audit bin. **AUDIT_UNPACK** Unpacks the compressed audit bin.
Buffer	Specifies the buffer containing the bin to be compressed or uncompressed. This buffer must contain a standard bin as described in the **audit.h** file.

Return Values

If the **auditpack** subroutine is successful, a pointer to a buffer containing the processed audit bin is returned. If unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditpack** subroutine fails if one or more of the following values is true:

Item	Description
EINVAL	The *Expand* parameter is not one of the valid values (**AUDIT_PACK** or **AUDIT_UNPACK**).
EINVAL	The *Expand* parameter is **AUDIT_UNPACK** and the packed data in *Buffer* does not unpack to its original size.
EINVAL	The *Expand* parameter is **AUDIT_PACK** and the bin in the *Buffer* parameter is already compressed, or the *Expand* parameter is **AUDIT_UNPACK** and the bin in the *Buffer* parameter is already unpacked.
ENOSPC	The **auditpack** subroutine is unable to allocate space for a new buffer.

auditproc Subroutine

Purpose

Gets or sets the audit state of a process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/audit.h>

int auditproc (ProcessID, Command, Argument, Length)
int  ProcessID;
int  Command;
char *  Argument;
int  Length;
```

Description

The **auditproc** subroutine queries or sets the auditing state of a process. There are two parts to the auditing state of a process:

- The list of classes to be audited for this process. Classes are defined by the **auditevents** subroutine. Each class includes a set of audit events. When a process causes an audit event, that event may be logged in the audit trail if it is included in one or more of the audit classes of the process.
- The audit status of the process. Auditing for a process may be suspended or resumed. Functions that generate an audit record can first check to see whether auditing is suspended. If process auditing is suspended, no audit events are logged for a process. For more information, see the **auditlog** subroutine.

Parameters

Item	Description
ProcessID	The process ID of the process to be affected. If *ProcessID* is 0, the **auditproc** subroutine affects the current process.
Command	The action to be taken. Defined in the **audit.h** file, valid values include: **AUDIT_KLIST_EVENTS** Sets the list of audit classes to be audited for the process and also sets the user's default audit classes definition within the kernel. The *Argument* parameter is a pointer to a list of null-terminated audit class names. The *Length* parameter is the length of this list, including null bytes. **AUDIT_QEVENTS** Returns the list of audit classes defined for the current process if *ProcessID* is 0. Otherwise, it returns the list of audit classes defined for the specified process ID. The *Argument* parameter is a pointer to a character buffer. The *Length* parameter specifies the size of this buffer. On return, this buffer contains a list of null-terminated audit class names. A null name terminates the list. **AUDIT_EVENTS** Sets the list of audit classes to be audited for the process. The *Argument* parameter is a pointer to a list of null-terminated audit class names. The *Length* parameter is the length of this list, including null bytes. **AUDIT_QSTATUS** Returns the audit status of the current process. You can only check the status of the current process. If the *ProcessID* parameter is nonzero, a -1 is returned and the **errno** global variable is set to **EINVAL**. The *Length* and *Argument* parameters are ignored. A return value of **AUDIT_SUSPEND** indicates that auditing is suspended. A return value of **AUDIT_RESUME** indicates normal auditing for this process. **AUDIT_STATUS** Sets the audit status of the current process. The *Length* parameter is ignored, and the *ProcessID* parameter must be zero. If *Argument* is **AUDIT_SUSPEND**, the audit status is set to suspend event auditing for this process. If the *Argument* parameter is **AUDIT_RESUME**, the audit status is set to resume event auditing for this process.
Argument	A character pointer for the audit class buffer for an **AUDIT_EVENT** or **AUDIT_QEVENTS** value of the *Command* parameter or an integer defining the audit status to be set for an **AUDIT_STATUS** operation.
Length	Size of the audit class character buffer.

Return Values

The **auditproc** subroutine returns the following values upon successful completion:

- The previous audit status (**AUDIT_SUSPEND** or **AUDIT_RESUME**), if the call queried or set the audit status (the *Command* parameter specified **AUDIT_QSTATUS** or **AUDIT_STATUS**)
- A value of 0 if the call queried or set audit events (the *Command* parameter specified **AUDIT_QEVENTS** or **AUDIT_EVENTS**)

Error Codes

If the **auditproc** subroutine fails if one or more of the following are true:

Item	Description
EINVAL	An invalid value was specified for the *Command* parameter.
EINVAL	The *Command* parameter is set to the **AUDIT_QSTATUS** or **AUDIT_STATUS** value and the **pid** value is nonzero.
EINVAl	The *Command* parameter is set to the **AUDIT_STATUS** value and the *Argument* parameter is not set to **AUDIT_SUSPEND** or **AUDIT_RESUME**.
ENOSPC	The *Command* parameter is **AUDIT_QEVENTS**, and the buffer size is insufficient. In this case, the first word of the *Argument* parameter is set to the required size.
EFAULT	The *Command* parameter is **AUDIT_QEVENTS** or **AUDIT_EVENTS** and the *Argument* parameter points to a location outside of the process' allocated address space.
ENOMEM	Memory allocation failed.
EPERM	The caller does not have root user authority.

auditread, auditread_r Subroutines

Purpose

Reads an audit record.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>
char *auditread ( FilePointer, AuditRecord)
FILE *FilePointer;
struct aud_rec *AuditRecord;


char *auditread_r ( FilePointer, AuditRecord,  RecordSize,  StreamInfo)
FILE *FilePointer;
struct aud_rec *AuditRecord;
size_t RecordSize;
void **StreamInfo;
```

Description

The **auditread** subroutine reads the next audit record from the specified file descriptor. Bins on this input stream are unpacked and uncompressed if necessary.

The **auditread** subroutine can not be used on more than one *FilePointer* as the results can be unpredictable. Use the **auditread_r** subroutine instead.

The **auditread_r** subroutine reads the next audit from the specified file descriptor. This subroutine is thread safe and can be used to handle multiple open audit files simultaneously by multiple threads of execution.

The **auditread_r** subroutine is able to read multiple versions of audit records. The version information contained in an audit record is used to determine the correct size and format of the record. When an input record header is larger than *AuditRecord*, an error is returned. In order to provide for binary compatibility

with previous versions, if *RecordSize* is the same size as the original (**struct aud_rec**), the input record is converted to the original format and returned to the caller.

Parameters

Item	Description
FilePointer	Specifies the file descriptor from which to read.
AuditRecord	Specifies the buffer to contain the header. The first short in this buffer must contain a valid number for the header.
RecordSize	The size of the buffer referenced by *AuditRecord*.
StreamInfo	A pointer to an opaque datatype used to hold information related to the current value of *FilePointer*. For each new value of *FilePointer*, a new *StreamInfo* pointer must be used. *StreamInfo* must be initialized to NULL by the user and is initialized by **auditread_r** when first used. When *FilePointer* has been closed, the value of *StreamInfo* can be passed to the **free** subroutine to be deallocated.

Return Values

If the **auditread** subroutine completes successfully, a pointer to a buffer containing the tail of the audit record is returned. The length of this buffer is returned in the `ah_length` field of the header file. If this subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditread** subroutine fails if one or more of the following is true:

Item	Description
EBADF	The *FilePointer* value is not valid.
ENOSPC	The **auditread** subroutine is unable to allocate space for the tail buffer.

Other error codes are returned by the **read** subroutine.

auditwrite Subroutine

Purpose

Writes an audit record.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/audit.h>
#include <stdio.h>
```

```
int auditwrite (Event, Result, Buffer1, Length1, Buffer2, Length2, ...)
char * Event;
int  Result;
char * Buffer1, *Buffer2 ...;
int  Length1, Length2 ...;
```

Description

The **auditwrite** subroutine builds the tail of an audit record and then writes it with the **auditlog** subroutine. The tail is built by gathering the specified buffers. The last buffer pointer must be a null.

If the **auditwrite** subroutine is to be called from a program invoked from the **initab** file, the **setpcred** subroutine should be called first to establish the process' credentials.

Parameters

Item	Description
Event	Specifies the name of the event to be logged.
Result	Specifies the audit status of the event. Valid values are defined in the **sys/audit.h** file and are listed in the **auditlog** subroutine.
Buffer1, Buffer2	Specifies the character buffers containing audit tail information. Note that numerical values must be passed by reference. The correct size can be computed with the **sizeof** C function.
Length1, Length2	Specifies the lengths of the corresponding buffers.

Return Values

If the **auditwrite** subroutine completes successfully, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **auditwrite** subroutine fails if the following is true:

Item	Description
ENOSPC	The **auditwrite** subroutine is unable to allocate space for the tail buffer.

Other error codes are returned by the **auditlog** subroutine.

authenticate Subroutine

Purpose

Verifies a user's name and password.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int authenticate (UserName, Response, Reenter, Message)
char *UserName;
char *Response;
int *Reenter;
char **Message;
```

Description

The **authenticate** subroutine maintains requirements users must satisfy to be authenticated to the system. It is a recallable interface that prompts for the user's name and password. The user must supply a character string at the prompt issued by the *Message* parameter. The *Response* parameter returns the user's response to the **authenticate** subroutine. The calling program makes no assumptions about the number of prompt messages the user must satisfy for authentication.

The *Reenter* parameter indicates when a user has satisfied all prompt messages. The parameter remains nonzero until a user has passed all prompts. After the returned value of *Reenter* is 0, the return code signals whether authentication has succeeded or failed. When progressing through prompts for a user, the value of *Reenter* must be maintained by the caller between invocations of **authenticate**.

The **authenticate** subroutine ascertains the authentication domains the user can attempt. The subroutine reads the **SYSTEM** line from the user's stanza in the **/etc/security/user** file. Each token that appears in the **SYSTEM** line corresponds to a method that can be dynamically loaded and processed. Likewise, the system can provide multiple or alternate authentication paths.

The **authenticate** routine maintains internal state information concerning the next prompt message presented to the user. If the calling program supplies a different user name before all prompts are complete for the user, the internal state information is reset and prompt messages begin again. The calling program maintains the value of the *Reenter* parameter while processing prompts for a given user.

If the user has no defined password, or the **SYSTEM** grammar explicitly specifies no authentication required, the user is not required to respond to any prompt messages. Otherwise, the user is always initially prompted to supply a password.

The **authenticate** subroutine can be called initially with the cleartext password in the *Response* parameter. If the user supplies a password during the initial invocation but does not have a password, authentication fails. If the user wants the **authenticate** subroutine to supply a prompt message, the *Response* parameter is a null pointer on initial invocation.

The **authenticate** subroutine sets the **AUTHSTATE** environment variable used by name resolution subroutines, such as the **getpwnam** subroutine. This environment variable indicates the registry to which to user authenticated. Values for the **AUTHSTATE** environment variable include **DCE**, **compat**, and token names that appear in a **SYSTEM** grammar. A null value can exist if the **cron** daemon or other utilities that do not require authentication is called.

Parameters

Item	Description
UserName	Points to the user's name that is to be authenticated.
Response	Specifies a character string containing the user's response to an authentication prompt.
Reenter	Points to a Boolean value that signals whether the **authenticate** subroutine has completed processing. If the *Reenter* parameter is a nonzero value, the **authenticate** subroutine expects the user to satisfy the prompt message provided by the *Message* parameter. If the *Reenter* parameter is 0, the **authenticate** subroutine has completed processing.
Message	Points to a pointer that the **authenticate** subroutine allocates memory for and fills in. This string is suitable for printing and issues prompt messages (if the *Reenter* parameter is a nonzero value). It also issues informational messages such as why the user failed authentication (if the *Reenter* parameter is 0). The calling application is responsible for freeing this memory.

Return Values

Upon successful completion, the **authenticate** subroutine returns a value of 0. If this subroutine fails, it returns a value of 1.

Error Codes

The **authenticate** subroutine is unsuccessful if one of the following values is true:

Item	Description
ENOENT	Indicates that the user is unknown to the system.
ESAD	Indicates that authentication is denied.
EINVAL	Indicates that the parameters are not valid.
ENOMEM	Indicates that memory allocation (malloc) failed.

Note: The DCE mechanism requires credentials on successful authentication that apply only to the authenticate process and its children.

authenticatex Subroutine

Purpose

Verifies a user's name and password.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int authenticatex (UserName, Response, Reenter, Message, State)
char *UserName;
char *Response;
int *Reenter;
char **Message;
void **State;
```

Description

The **authenticatex** subroutine maintains requirements that users must satisfy to be authenticated to the system. It is a recallable interface that prompts for the user's name and password. The user must supply a character string at the prompt issued by the *Message* parameter. The *Response* parameter returns the user's response to the **authenticatex** subroutine. The calling program makes no assumptions about the number of prompt messages the user must satisfy for authentication. The **authenticatex** subroutine maintains information about the results of each part of the authentication process in the *State* parameter. This parameter can be shared with the **chpassx**, **loginrestrictionsx** and **passwdexpiredx** subroutines. The proper sequence of library routines for authenticating a user in order to create a new session is:

1. Call the **loginrestrictionsx** subroutine to determine which administrative domains allow the user to log in.
2. Call the **authenticatex** subroutine to perform authentication using those administrative domains that grant login access.
3. Call the **passwdexpiredx** subroutine to determine if any of the passwords used during the authentication process have expired and must be changed in order for the user to be granted access.
4. If the **passwdexpiredx** subroutine indicated that one or more passwords have expired and must be changed by the user, call the **chpassx** subroutine to update all of the passwords that were used for the authentication process.

The *Reenter* parameter remains a nonzero value until the user satisfies all prompt messages or answers incorrectly. When the *Reenter* parameter is 0, the return code signals whether authentication passed or failed. The value of the *Reenter* parameter must be 0 on the initial call. A nonzero value for the *Reenter* parameter must be passed to the **authenticatex** subroutine on subsequent calls. A new authentication can be begun by calling the **authenticatex** subroutine with a 0 value for the *Reenter* parameter or by using a different value for *UserName*.

The *State* parameter contains information about the authentication process. The *State* parameter from an earlier call to **loginrestrictionsx** can be used to control how authentication is performed. Administrative domains that do not permit the user to log in cause those administrative domains to be ignored during authentication even if the user has the correct authentication information.

The **authenticatex** subroutine ascertains the authentication domains the user can attempt. The subroutine uses the **SYSTEM** attribute for the user. Each token that is displayed in the **SYSTEM** line corresponds to a method that can be dynamically loaded and processed. Likewise, the system can provide multiple or alternate authentication paths.

The **authenticatex** subroutine maintains internal state information concerning the next prompt message presented to the user. If the calling program supplies a different user name before all prompts are complete for the user, the internal state information is reset and prompt messages begin again. The **authenticatex** subroutine requires that the *State* parameter be initialized to reference a null value when changing user names or that the *State* parameter from an earlier call to **loginrestrictionsx** for the new user be provided.

If the user has no defined password, or the **SYSTEM** grammar explicitly specifies no authentication required, the user is not required to respond to any prompt messages. Otherwise, the user is always initially prompted to supply a password.

The **authenticatex** subroutine can be called initially with the `cleartext` password in the *Response* parameter. If the user supplies a password during the initial invocation but does not have a password, authentication fails. If the user wants the **authenticatex** subroutine to supply a prompt message, the *Response* parameter is a null pointer on initial invocation.

The **authenticatex** subroutine sets the **AUTHSTATE** environment variable used by name resolution subroutines, such as the **getpwnam** subroutine. This environment variable indicates the first registry to which the user authenticated. Values for the **AUTHSTATE** environment variable include **DCE**, **compat**, and token names that appear in a **SYSTEM** grammar. A null value can exist if the **cron** daemon or another utility that does not require authentication is called.

Parameters

Item	Description
Message	Points to a pointer that the **authenticatex** subroutine allocates memory for and fills in. This string is suitable for printing and issues prompt messages (if the *Reenter* parameter is a nonzero value). It also issues informational messages, such as why the user failed authentication (if the *Reenter* parameter is 0). The calling application is responsible for freeing this memory.
Reenter	Points to an integer value that signals whether the **authenticatex** subroutine has completed processing. If the integer referenced by the *Reenter* parameter is a nonzero value, the **authenticatex** subroutine expects the user to satisfy the prompt message provided by the *Message* parameter. If the integer referenced by the *Reenter* parameter is 0, the **authenticatex** subroutine has completed processing. The initial value of the integer referenced by *Reenter* must be 0 when the **authenticatex** function is initially invoked and must not be modified by the calling application until the **authenticationx** subroutine has completed processing.
Response	Specifies a character string containing the user's response to an authentication prompt.

Item	Description
State	Points to a pointer that the **authenticatex** subroutine allocates memory for and fills in. The *State* parameter can also be the result of an earlier call to the **loginrestrictionsx** subroutine. This parameter contains information about the results of the authentication process for each term in the user's **SYSTEM** attribute. The calling application is responsible for freeing this memory when it is no longer needed for a subsequent call to the **passwdexpiredx** or **chpassx** subroutines.
UserName	Points to the user's name that is to be authenticated.

Return Values

Upon successful completion, the **authenticatex** subroutine returns a value of 0. If this subroutine fails, it returns a value of 1.

Error Codes

The **authenticatex** subroutine is unsuccessful if one of the following values is true:

Item	Description
EINVAL	The parameters are not valid.
ENOENT	The user is unknown to the system.
ENOMEM	Memory allocation (malloc) failed.
ESAD	Authentication is denied.

Note: Additional information about the behavior of a loadable authentication module can be found in the documentation for that module.

b

The following Base Operating System (BOS) runtime services begin with the letter *b*.

basename Subroutine

Purpose

Return the last element of a path name.

Library

Standard C Library **(libc.a)**

Syntax

#include <libgen.h>

char *basename (char **path*)

Description

Given a pointer to a character string that contains a path name, the **basename** subroutine deletes trailing "/" characters from *path*, and then returns a pointer to the last component of *path*. The "/" character is defined as trailing if it is not the first character in the string.

If *path* is a null pointer or points to an empty string, a pointer to a static constant "." is returned.

Return Values

The **basename** function returns a pointer to the last component of *path*.

The **basename** function returns a pointer to a static constant "." if *path* is a null pointer or points to an empty string.

The **basename** function may modify the string pointed to by *path* and may return a pointer to static storage that may then be overwritten by a subsequent call to the **basename** subroutine.

Examples

Input string	Output string
"/usr/lib"	"lib"
"/usr/"	"usr"
"/"	"/"

baudrate Subroutine

Purpose

Gets the terminal baud rate.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
int baudrate(void)
```

Description

The **baudrate** subroutine extracts the output speed of the terminal in bits per second.

Return Values

The **baudrate** subroutine returns the output speed of the terminal.

Examples

To query the baud rate and place the value in the user-defined integer variable `BaudRate`, enter:

```
BaudRate = baudrate();
```

bcopy, bcmp, bzero, ffs, ffsl, or ffsll Subroutine

Purpose

Performs bit and byte string operations.

Library

Standard C Library (**libc.a**)

Syntax

#include <strings.h>

void bcopy (*Source***,** *Destination***,** *Length***) const void ****Source***, char ****Destination***; size_t** *Length***;**

int bcmp (*String1***,** *String2***,** *Length***) const void ****String1***, ****String2***; size_t** *Length***;**

void bzero (*String***,***Length***) char ****String***; int** *Length***;**

int ffs (*Index***) int** *Index***;**

int ffsl(*Index*) **longint***Index*;

int ffsll(*Index*)**longlongint***Index*;

Description

Note: The **bcopy** subroutine takes parameters backwards from the **strcpy** subroutine.

The **bcopy**, **bcmp**, and **bzero** subroutines operate on variable length strings of bytes. They do not check for null bytes as do the **string** routines.

The **bcopy** subroutine copies the value of the *Length* parameter in bytes from the string in the *Source* parameter to the string in the *Destination* parameter.

The **bcmp** subroutine compares the byte string in the *String1* parameter against the byte string of the *String2* parameter, returning a zero value if the two strings are identical and a nonzero value otherwise. Both strings are assumed to be *Length* bytes long.

The **bzero** subroutine zeroes out the string in the *String* parameter for the value of the *Length* parameter in bytes.

The **ffs** subroutine finds the first bit set in the *Index* parameter passed to it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates that the value passed is 0. The least significant bit is position 1 and the most significant position is 32 or 64.

The **ffsl()** and **ffsll()** subroutines perform the same function as the **ffs** subroutine for arguments of different sizes.

beep Subroutine

Purpose

Sounds the audible alarm on the terminal.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int beep(void);
```

Description

The **beep** subroutine alerts the user. It sounds the audible alarm on the terminal, or if that is not possible, it flashes the screen (visible bell). If neither signal is possible, nothing happens.

Return Values

The **beep** subroutine always returns OK.

Examples

To sound an audible alarm, enter:

```
beep();
```

bessel: j0, j1, jn, y0, y1, or yn Subroutine

Purpose

Computes Bessel functions.

Libraries

```
IEEE Math Library (libm.a)
or System V Math Library (libmsaa.a)
```

Syntax

```
#include <math.h>
```

```
double j0 (x)
double  x;
```

```
double j1 (x)
double x;
```

```
double jn (n, x)
int  n;
double x;
```

```
double y0 (x)
double x;
```

```
double y1 (x)
double x;
```

```
double yn (n, x)
int n;
double x;
```

Description

Bessel functions are used to compute wave variables, primarily in the field of communications.

The **j0** subroutine and **j1** subroutine return Bessel functions of *x* of the first kind, of orders 0 and 1, respectively. The **jn** subroutine returns the Bessel function of *x* of the first kind of order *n*.

The **y0** subroutine and **y1** subroutine return the Bessel functions of *x* of the second kind, of orders 0 and 1, respectively. The **yn** subroutine returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **-lm** flag. To compile the **j0.c** file, for example:

```
cc j0.c -lm
```

Parameters

Item	Description
x	Specifies some double-precision floating-point value.
n	Specifies some integer value.

Return Values

When using **libm.a** (**-lm**), if *x* is negative, **y0**, **y1**, and **yn** return the value NaNQ. If *x* is 0, **y0**, **y1**, and **yn** return the value **-HUGE_VAL**.

When using **libmsaa.a** (**-lmsaa**), values too large in magnitude cause the functions **j0**, **j1**, **y0**, and **y1** to return 0 and to set the **errno** global variable to ERANGE. In addition, a message indicating TLOSS error is printed on the standard error output.

Nonpositive values cause **y0**, **y1**, and **yn** to return the value **-HUGE** and to set the **errno** global variable to **EDOM**. In addition, a message indicating argument DOMAIN error is printed on the standard error output.

These error-handling procedures may be changed with the **matherr** subroutine when using **libmsaa.a** (**-lmsaa**).

bindprocessor Subroutine

Purpose

Binds kernel threads to a processor.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/processor.h>

int bindprocessor ( What,  Who,  Where)
int What;
int Who;
cpu_t Where;
```

Description

The **bindprocessor** subroutine binds a single kernel thread, or all kernel threads in a process, to a processor, forcing the bound threads to be scheduled to run on that processor. It is important to understand that a process itself is not bound, but rather its kernel threads are bound. Once kernel threads are bound, they are always scheduled to run on the chosen processor, unless they are later unbound. When a new thread is created, it has the same bind properties as its creator. This applies to the initial thread in the new process created by the **fork** subroutine: the new thread inherits the bind properties of the thread which called **fork**. When the **exec** subroutine is called, thread properties are left unchanged.

The **bindprocessor** subroutine will fail if the target process has a *Resource Attachment*.

Programs that use processor bindings should become Dynamic Logical Partitioning (DLPAR) aware.

Parameters

Item	Description
What	Specifies whether a process or a thread is being bound to a processor. The *What* parameter can take one of the following values: **BINDPROCESS** A process is being bound to a processor. **BINDTHREAD** A thread is being bound to a processor.
Who	Indicates a process or thread identifier, as appropriate for the *What* parameter, specifying the process or thread which is to be bound to a processor.
Where	If the *Where* parameter is a bind CPU identifier, it specifies the processor to which the process or thread is to be bound. A value of **PROCESSOR_CLASS_ANY** unbinds the specified process or thread, which will then be able to run on any processor. The **sysconf** subroutine can be used to retrieve information about the number of online processors in the system.

Return Values

On successful completion, the **bindprocessor** subroutine returns 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **bindprocessor** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	The *What* parameter is invalid, or the *Where* parameter indicates an invalid processor number or a processor class which is not currently available.
ESRCH	The specified process or thread does not exist.
EPERM	The caller does not have root user authority, and the *Who* parameter specifies either a process, or a thread belonging to a process, having a real or effective user ID different from that of the calling process. The target process has a *Resource Attachment*.

box Subroutine

Purpose

Draws borders from single-byte characters and renditions.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int box(WINDOW  *win,
chtype  verch,
chtype  horch);
```

Description

The **box** subroutine draws a border around the edges of the specified window. This subroutine does not advance the cursor position. This subroutine does not perform special character processing or perform wrapping.

The **box** subroutine (**win*, *verch*, *horch*) has an effect equivalent to:

```
wborder(win, verch, verch, horch, horch, 0, 0, 0, 0);
```

Parameters

Item	Description
horch	Specifies the character to draw the horizontal lines of the box. The character must be a 1-column character.
verch	Specifies the character to draw the vertical lines of the box. The character must be a 1-column character.
**win*	Specifies the window to draw the box in or around.

Return Values

Upon successful completion, the **box** function returns OK. Otherwise, it returns ERR.

Examples

1. To draw a box around the user-defined window, `my_window`, using | (pipe) as the vertical character and - (minus sign) as the horizontal character, enter:

```
WINDOW *my_window;
box(my_window, '|', '-');
```

2. To draw a box around my_window using the default characters ACS_VLINE and ACS_HLINE, enter:

```
WINDOW *my_window;
box(my_window, 0, 0);
```

brk or sbrk Subroutine

Purpose

Changes data segment space allocation.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd .h>
```

int brk (*EndDataSegment*)
char **EndDataSegment*;

void *sbrk (*Increment*)
intptr_t *Increment*;

Description

The **brk** and **sbrk** subroutines dynamically change the amount of space allocated for the data segment of the calling process. (For information about segments, see the **exec** subroutine. For information about the maximum amount of space that can be allocated, see the **ulimit** and **getrlimit** subroutines.)

The change is made by resetting the break value of the process, which determines the maximum space that can be allocated. The break value is the address of the first location beyond the current end of the data region. The amount of available space increases as the break value increases. The available space is initialized to a value of 0 at the time it is used. The break value can be automatically rounded up to a size appropriate for the memory management architecture.

The **brk** subroutine sets the break value to the value of the *EndDataSegment* parameter and changes the amount of available space accordingly.

The **sbrk** subroutine adds to the break value the number of bytes contained in the *Increment* parameter and changes the amount of available space accordingly. The *Increment* parameter can be a negative number, in which case the amount of available space is decreased.

Parameters

Item	Description
EndDataSegment	Specifies the effective address of the maximum available data.
Increment	Specifies any integer.

Return Values

Upon successful completion, the **brk** subroutine returns a value of 0, and the **sbrk** subroutine returns the old break value. If either subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **brk** subroutine and the **sbrk** subroutine are unsuccessful and the allocated space remains unchanged if one or more of the following are true:

Item	Description
ENOMEM	The requested change allocates more space than is allowed by a system-imposed maximum. (For information on the system-imposed maximum on memory space, see the **ulimit** system call.)
ENOMEM	The requested change sets the break value to a value greater than or equal to the start address of any attached shared-memory segment. (For information on shared memory operations, see the **shmat** subroutine.)

bsearch Subroutine

Purpose

Performs a binary search.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

void *bsearch (*Key*, *Base*, *NumberOfElements*, *Size*, *ComparisonPointer*)

```
const void *Key;
const void *Base;
size_t NumberOfElements;
size_t Size;
int (*ComparisonPointer) (const void *, const void *);
```

Description

The **bsearch** subroutine is a binary search routine.

The **bsearch** subroutine searches an array of *NumberOfElements* objects, the initial member of which is pointed to by the *Base* parameter, for a member that matches the object pointed to by the *Key* parameter. The size of each member in the array is specified by the *Size* parameter.

The array must already be sorted in increasing order according to the provided comparison function *ComparisonPointer* parameter.

Parameters

Item	Description
Key	Points to the object to be sought in the array.

Item	Description
Base	Points to the element at the base of the table.
NumberOfElements	Specifies the number of elements in the array.
ComparisonPointer	Points to the comparison function, which is called with two arguments that point to the *Key* parameter object and to an array member, in that order.
Size	Specifies the size of each member in the array.

Return Values

If the *Key* parameter value is found in the table, the **bsearch** subroutine returns a pointer to the element found.

If the *Key* parameter value is not found in the table, the **bsearch** subroutine returns the null value. If two members compare as equal, the matching member is unspecified.

For the *ComparisonPointer* parameter, the comparison function compares its parameters and returns a value as follows:

- If the first parameter is less than the second parameter, the *ComparisonPointer* parameter returns a value less than 0.
- If the first parameter is equal to the second parameter, the *ComparisonPointer* parameter returns a value of 0.
- If the first parameter is greater than the second parameter, the *ComparisonPointer* parameter returns a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

The *Key* and *Base* parameters should be of type pointer-to-element and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

btowc Subroutine

Purpose

Single-byte to wide-character conversion.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>
```

```
wint_t btowc (intc);
```

Description

The *btowc* function determines whether c constitutes a valid (one-byte) character in the initial shift state.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

The btowc function returns WEOF if c has the value EOF or if (unsigned char) c does not constitute a valid (one-byte) character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

buildproclist Subroutine

Purpose

Retrieves a list of process transaction records based on the criteria specified.

Library

The `libaacct.a` library.

Syntax

```
#define <sys/aacct.h>
int buildproclist(crit, crit_list, n_crit, p_list, sublist)
int crit;
union proc_crit *crit_list;
int n_crit;
struct aacct_tran_rec *p_list;
struct aacct_tran_rec **sublist;
```

Description

The `buildproclist` subroutine retrieves a subset of process transaction records from the master process transaction records that are given as input based on the selection criteria provided. This selection criteria can be one of the following values defined in `sys/aacct.h`:

- `CRIT_UID`
- `CRIT_GID`
- `CRIT_PROJ`
- `CRIT_CMD`

For example, if the criteria is specified as `CRIT_UID`, the list of process transaction records for specific user IDs will be retrieved. The list of user IDs are passed through the *crit_list* argument of type `union proc_crit`. Based on the specified criteria, the caller has to pass an array of user IDs, group IDs, project IDs or command names in this union.

Usually, the master list of transaction records is obtained by a prior call to the `getproclist` subroutine.

Parameters

Item	Description
crit	Integer value representing the selection criteria for the process records.
crit_list	Pointer to `union proc_crit` where the data for the selection criteria is passed.
n_crit	Number of elements to be considered for the selection, such as the number of user IDs.
p_list	Master list of process transaction records.
sublist	Pointer to the linked list of `aacct_tran_rec` structures, which hold the retrieved process transaction records.

Security

No restrictions. Any user can call this function.

Return Values

Item	Description
0	The call to the subroutine was successful.
-1	The call to the subroutine failed.

Error Codes

Item	Description
EINVAL	The passed pointer is NULL.
ENOMEM	Insufficient memory.
EPERM	Permission denied. Unable to read the data file.

buildtranlist or freetranlist Subroutine

Purpose

Read the advanced accounting records from the advanced accounting data file.

Library

The libaacct.a library.

Syntax

```
#define <sys/aacct.h>
buildtranlist(filename, trid[], ntrids, begin_time, end_time, tran_list)
char *filename;
unsigned int trid[];
unsigned int ntrids;
long long begin_time;
long long end_time;
struct aacct_tran_rec **tran_list;
freetranlist(tran_list)
struct aacct_tran_rec *tran_list;
```

Description

The buildtranlist subroutine retrieves the transaction records of the specified transaction type from the accounting data file. The required transaction IDs are passed as arguments, and these IDs are defined in sys/aacct.h. The list of transaction records are returned to the calling program through the *tran_list* pointer argument.

This API can be called multiple times with different accounting data file names to generate a consolidated list of transaction records from multiple data files. It appends the new file data to the end of the linked list pointed to by the *tran_list* argument. In addition, it internally sorts the transaction records based on the time of transaction so users can get a time-sorted list of transaction records from this routine. This subroutine can also be used to retrieve the intended transaction records for a particular interval of time by specifying the begin and end times of this interval as arguments.

The freetranlist subroutine frees the memory allocated to these transaction records. It can be used to deallocate memory that has been allocated to the transaction record lists created by routines such as buildtranlist, getproclist, getlparlist, and getarmlist.

Parameters

Item	Description
begin_time	Specifies the start timestamp for collecting records in a particular intervals. The input is in seconds since EPOCH. Specifying `-1` retrieves all the records.
end_time	Specifies the end timestamp for collecting records in a particular intervals. The input is in seconds since EPOCH. Specifying `-1` retrieves all the records.
filename	Name of the advanced accounting data file.
ntrids	Count of transaction IDs passed in the array *trid*.
tran_list	Pointer to the linked list of `aacct_tran_rec` structures that are to be returned to the caller or freed.
trid	An array of transaction record type identifiers.

Security

No restrictions. Any user can call this function.

Return Values

Item	Description
0	The call to the subroutine was successful.
-1	The call to the subroutine failed.

Error Codes

Item	Description
`EINVAL`	The passed pointer is NULL.
`ENOENT`	Specified data file does not exist.
`ENOMEM`	Insufficient memory.
`EPERM`	Permission denied. Unable to read the data file.

c

The following Base Operating System (BOS) runtime services begin with the letter *c*.

_check_lock Subroutine

Purpose

Conditionally updates a single word variable atomically.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/atomic_op.h>
```

```
boolean_t _check_lock ( word_addr,  old_val,  new_val)
atomic_p word_addr;
int old_val;
int new_val;
```

Parameters

Item	Description
word_addr	Specifies the address of the single word variable.
old_val	Specifies the old value to be checked against the value of the single word variable.
new_val	Specifies the new value to be conditionally assigned to the single word variable.

Description

The **_check_lock** subroutine performs an atomic (uninterruptible) sequence of operations. The **compare_and_swap** subroutine is similar, but does not issue synchronization instructions and therefore is inappropriate for updating lock words.

Note: The word variable must be aligned on a full word boundary.

Return Values

Item	Description
FALSE	Indicates that the single word variable was equal to the old value and has been set to the new value.
TRUE	Indicates that the single word variable was not equal to the old value and has been left unchanged.

_clear_lock Subroutine

Purpose

Stores a value in a single word variable atomically.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/atomic_op.h>

void _clear_lock ( word_addr,  value)
atomic_p word_addr;
int value
```

Parameters

Item	Description
word_addr	Specifies the address of the single word variable.
value	Specifies the value to store in the single word variable.

Description

The **_clear_lock** subroutine performs an atomic (uninterruptible) sequence of operations.

This subroutine has no return values.

Note: The word variable must be aligned on a full word boundary.

cabs, cabsf, or cabsl Subroutine

Purpose

Returns a complex absolute value.

Syntax

```
#include <complex.h>

double cabs (z)
double complex z;

float cabsf (z)
float complex z;

long double cabsl (z)
long double complex z;
```

Description

The **cabs**, **cabsf**, or **cabsl** subroutines compute the complex absolute value (also called norm, modulus, or magnitude) of the *z* parameter.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

Returns the complex absolute value.

cacos, cacosf, or cacosl Subroutine

Purpose

Computes the complex arc cosine.

Syntax

```
#include <complex.h>

double complex cacos (z)
double complex z;

float complex cacosf (z)
float complex z;

long double complex cacosl (z)
long double complex z;
```

Description

The **cacos**, **cacosf**, or **cacosl** subroutine computes the complex arc cosine of *z*, with branch cuts outside the interval [–1, +1] along the real axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **cacos**, **cacosf**, or **cacosl** subroutine returns the complex arc cosine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [0, pi] along the real axis.

cacosh, cacoshf, or cacoshl Subroutines

Purpose

Computes the complex arc hyperbolic cosine.

Syntax

```
#include <complex.h>

double complex cacosh (z)
double complex z;

float complex cacoshf (z)
float complex z;

long double complex cacoshl (z)
long double complex z;
```

Description

The **cacosh**, **cacoshf**, or **cacoshl** subroutine computes the complex arc hyperbolic cosine of the *z* parameter, with a branch cut at values less than 1 along the real axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **cacosh**, **cacoshf**, or **cacoshl** subroutine returns the complex arc hyperbolic cosine value, in the range of a half-strip of non-negative values along the real axis and in the interval [-*i* pi , +*i* pi] along the imaginary axis.

call_once Subroutine

Purpose

Runs the function that is specified by the *func* parameter only once, even if the function is called from several threads.

Library

Standard C library (**libc.a**)

Syntax

```
#include <threads.h>
void call_once (once_flag * flag void * func (void));
```

Description

The **call_once** subroutine uses the *once_flag* value specified by the **flag** parameter to ensure that the function specified by the **func** parameter is called exactly once when the **call_once** subroutine is called for the first time, with the value of the **flag** parameter.

An effective call to the **call_once** subroutine synchronizes all the subsequent calls to the **call_once** subroutine by using the same value of the **flag** parameter.

Parameters

Item	Description
flag	Specifies the value of the parameter to call the **call_once** subroutine and to synchronize all further calls with this flag.
func	Specifies the function that is called only once.

Return Values

No return value.

Files

The **threads.h** file defines standard macros, data types, and subroutines.

can_change_color, color_content, has_colors,init_color, init_pair, start_color or pair_content Subroutine

Purpose

Color manipulation functions and external variables for color support.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

bool can_change_color(void);

int color_content(short color,
short *red,
short *green,
short *blue);

int COLOR_PAIR(int n);

bool has_colors(void);

int init_color
(short color,
short red,
short green,
short blue);

int init_pair
(short pair,
short f,
short b);

int pair_content
(short pair,
short *f,
short *b);

int PAIR_NUMBER
(int value);
int start_color
(void);

extern int COLOR_PAIRS;
extern int COLORS;
```

Description

These functions manipulate color on terminals that support color.

Querying Capabilities

The **has_colors** subroutine indicates whether the terminal is a color terminal. The **can_change_color** subroutine indicates whether the terminal is a color terminal on which colors can be redefined.

Initialisation

The **start_color** subroutine must be called in order to enable use of colors and before any color manipulation function is called. This subroutine initializes eight basic colors (black, blue, green, cyan, red, magenta, yellow, and white) that can be specified by the color macros (such as **COLOR_BLACK**) defined in <**curses.h**>. The initial appearance of these eight colors is not specified.

The function also initialises two global external variables:

- **COLORS** defines the number of colors that the terminal supports. If **COLORS** is 0, the terminal does not support redefinition of colors (and **can_change_color** subroutine will return **FALSE**).
- **COLOR_PAIRS** defines the maximum number of color-pairs that the terminal supports.

Color Identification

The **init_color** subroutine redefines color number color, on terminals that support the redefinition of colors, to have the red, green, and blue intensity components specified by red, green, and blue, respectively. Calling **init_color** subroutine also changes all occurrences of the specified color on the screen to the new definition.

The **color_content** subroutine identifies the intensity components of color number color. It stores the red, green, and blue intensity components of this color in the addresses pointed to by red, green, and blue, respectively.

For both functions, the color argument must be in the range from **0** to and including **COLORS -1**. Valid intensity values range from **0** (no intensity component) up to and including **1000** (maximum intensity in that component).

User-Defined Color Pairs

Calling **init_pair** defines or redefines color-pair number pair to have foreground color f and background color b. Calling **init_pair** changes any characters that were displayed in the color pair's old definition to the new definition and refreshes the screen.

After defining the color pair, the macro **COLOR_PAIR**(n) returns the value of color pair n. This value is the color attribute as it would be extracted from a **chtype**. Conversely, the macro **PAIR_NUMBER**(value) returns the color pair number associated with the color attribute value.

The **pair_content** subroutine retrieves the component colors of a color-pair number pair. It stores the foreground and background color numbers in the variables pointed to by *f* and *b*, respectively.

With **init_pair** and **pair_content** subroutines, the value of pair must be in a range from **0** to and including **COLOR_PAIRS -1**. (There may be an implementation-specific upper limit on the valid value of pair, but any such limit is at least 63.) Valid values for *f* and *b* are the range from **0** to and including **COLORS -1**.

The **can_change_color** subroutine returns TRUE if the terminal supports colors and can change their definitions; otherwise, it returns FALSE.

Parameters

Item	Description
color	
**red*	
**green*	
**blue*	
pair	
f	
b	
value	

Return Values

The **has_colors** subroutine returns TRUE if the terminal can manipulate colors; otherwise, it returns FALSE.

Upon successful completion, the other functions return OK. Otherwise, they return ERR.

Examples

For the **can_change_color** subroutine:

To test whether or not a terminal can change its colors, enter the following and check the return for TRUE or FALSE:

```
can_change_color();
```

For the **color_content** subroutine:

To obtain the RGB component information for color 10 (assuming the terminal supports at least 11 colors), use:

```
short *r, *g, *b;
color_content(10,r,g,b);
```

For the **has_color** subroutine:

To determine whether or not a terminal supports color, use:

```
has_colors();
```

For the **pair_content** subroutine:

To obtain the foreground and background colors for color-pair 5, use:

```
short *f, *b;
pair_content(5,f,b);
```

For this subroutine to succeed, you must have already initialized the color pair. The foreground and background colors will be stored at the locations pointed to by *f* and *b*.

For the **start_color** subroutine:

To enable the color support for a terminal that supports color, use:

```
start_color();
```

For the **init_pair** subroutine:

To initialize the color definition for color-pair 2 to a black foreground (color 0) with a cyan background (color 3), use:

```
init_pair(2,COLOR_BLACK, COLOR_CYAN);
```

For the **init_color** subroutine:

To initialize the color definition for color 11 to violet on a terminal that supports at least 12 colors, use:

```
init_color(11,500,0,500);
```

carg, cargf, or cargl Subroutine

Purpose

Returns the complex argument value.

Syntax

```
#include <complex.h>

double carg (z)
double complex z;
```

```
float cargf (z)
float complex z;

long double cargl (z)
long double complex z;
```

Description

The **carg**, **cargf**, or **cargl** subroutine computes the argument (also called phase angle) of the *z* parameter, with a branch cut along the negative real axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **carg**, **cargf**, or **cargl** subroutine returns the value of the argument in the interval [-pi , +pi].

casin, casinf, or casinl Subroutine

Purpose

Computes the complex arc sine.

Syntax

```
#include <complex.h>

double complex casin (z)
double complex z;

float complex casinf (z)
float complex z;

long double complex casinl (z)
long double complex z;
```

Description

The **casin**, **casinf**, or **casinl** subroutine computes the complex arc sine of the *z* parameter, with branch cuts outside the interval [–1, +1] along the real axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **casin**, **casinf**, or **casinl** subroutine returns the complex arc sine value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

casinh, casinfh, or casinlh Subroutine

Purpose

Computes the complex arc hyperbolic sine.

Syntax

```
#include <complex.h>

double complex casinh (z)
double complex z;

float complex casinhf (z)
float complex z;

long double complex casinhl (z)
long double complex z;
```

Description

The **casinh**, **casinfh**, and **casinlh** subroutines compute the complex arc hyperbolic sine of the *z* parameter, with branch cuts outside the interval [*-i, +i*] along the imaginary axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **casinh**, **casinfh**, and **casinlh** subroutines return the complex arc hyperbolic sine value, in the range of a strip mathematically unbounded along the real axis and in the interval [*-i* pi/2, *+i* pi/2] along the imaginary axis.

catan, catanf, or catanl Subroutine

Purpose

Computes the complex arc tangent.

Syntax

```
#include <complex.h>

double complex catan (z)
double complex z;

float complex catanf (z)
float complex z;

long double complex catanl (z)
long double complex z;
```

Description

The **catan**, **catanf**, and **catanl** subroutines compute the complex arc tangent of *z*, with branch cuts outside the interval [*-i*, *+i*] along the imaginary axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **catan**, **catanf**, and **catanl** subroutines return the complex arc tangent value, in the range of a strip mathematically unbounded along the imaginary axis and in the interval [-pi/2, +pi/2] along the real axis.

catanh, catanhf, or catanhl Subroutine

Purpose

Computes the complex arc hyperbolic tangent.

Syntax

```
#include <complex.h>

double complex catanh (z)
double complex z;

float complex catanhf (z)
float complex z;

long double complex catanhl (z)
long double complex z;
```

Description

The **catanh**, **catanhf**, and **catanhl** subroutines compute the complex arc hyperbolic tangent of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **catanh**, **catanhf**, and **catanhl** subroutines return the complex arc hyperbolic tangent value, in the range of a strip mathematically unbounded along the real axis and in the interval [-*i* pi/2, +*i* pi/2] along the imaginary axis.

catclose Subroutine

Purpose

Closes a specified message catalog.

Library

Standard C Library (**libc.a**)

Syntax

#include <nl_types.h>

```
int catclose ( CatalogDescriptor)
nl_catd CatalogDescriptor;
```

Description

The **catclose** subroutine closes a specified message catalog. If your program accesses several message catalogs and you reach the maximum number of opened catalogs (specified by the **NL_MAXOPEN** constant), you must close some catalogs before opening additional ones. If you use a file descriptor to implement the **nl_catd** data type, the **catclose** subroutine closes that file descriptor.

The **catclose** subroutine closes a message catalog only when the number of calls it receives matches the total number of calls to the **catopen** subroutine in an application. All message buffer pointers obtained by prior calls to the **catgets** subroutine are not valid when the message catalog is closed.

Parameters

Item	Description
CatalogDescriptor	Points to the message catalog returned from a call to the **catopen** subroutine.

Return Values

The **catclose** subroutine returns a value of 0 if it closes the catalog successfully, or if the number of calls it receives is fewer than the number of calls to the **catopen** subroutine.

The **catclose** subroutine returns a value of -1 if it does not succeed in closing the catalog. The **catclose** subroutine is unsuccessful if the number of calls it receives is greater than the number of calls to the **catopen** subroutine, or if the value of the *CatalogDescriptor* parameter is not valid.

catgets Subroutine

Purpose

Retrieves a message from a catalog.

Library

Standard C Library (**libc.a**)

Syntax

#include <nl_types>

```
char *catgets (CatalogDescriptor, SetNumber, MessageNumber, String)
nl_catd  CatalogDescriptor;
int  SetNumber,  MessageNumber;
const char * String;
```

Description

The **catgets** subroutine retrieves a message from a catalog after a successful call to the **catopen** subroutine. If the **catgets** subroutine finds the specified message, it loads it into an internal character string buffer, ends the message string with a null character, and returns a pointer to the buffer.

The **catgets** subroutine uses the returned pointer to reference the buffer and display the message. However, the buffer can not be referenced after the catalog is closed.

Parameters

Item	Description
CatalogDescriptor	Specifies a catalog description that is returned by the **catopen** subroutine.
SetNumber	Specifies the set ID.
MessageNumber	Specifies the message ID. The *SetNumber* and *MessageNumber* parameters specify a particular message to retrieve in the catalog.
String	Specifies the default character-string buffer.

Return Values

If the **catgets** subroutine is unsuccessful for any reason, it returns the user-supplied default message string specified by the *String* parameter.

catopen Subroutine

Purpose

Opens a specified message catalog.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <nl_types.h>

nl_catd catopen ( CatalogName,  Parameter)
const char *CatalogName;
int Parameter;
```

Description

The **catopen** subroutine opens a specified message catalog and returns a catalog descriptor used to retrieve messages from the catalog. The contents of the catalog descriptor are complete when the **catgets** subroutine accesses the message catalog. The **nl_catd** data type is used for catalog descriptors and is defined in the **nl_types.h** file.

If the catalog file name referred to by the *CatalogName* parameter contains a leading / (slash), it is assumed to be an absolute path name. If the catalog file name is not an absolute path name, the user environment determines which directory paths to search. The **NLSPATH** environment variable defines the directory search path. When this variable is used, the **setlocale** subroutine must be called before the **catopen** subroutine.

A message catalog descriptor remains valid in a process until that process or a successful call to one of the **exec** functions closes it.

You can use two special variables, **%N** and **%L**, in the **NLSPATH** environment variable. The **%N** variable is replaced by the catalog name referred to by the call that opens the message catalog. The **%L** variable is replaced by the value of the **LC_MESSAGES** category.

The value of the **LC_MESSAGES** category can be set by specifying values for the **LANG**, **LC_ALL**, or **LC_MESSAGES** environment variable. The value of the **LC_MESSAGES** category indicates which locale-specific directory to search for message catalogs. For example, if the **catopen** subroutine specifies a catalog with the name `mycmd`, and the environment variables are set as follows:

```
NLSPATH=../%N:./%N:/system/nls/%L/%N:/system/nls/%N LANG=fr_FR
```

then the application searches for the catalog in the following order:

```
../mycmd
./mycmd
/system/nls/fr_FR/mycmd
/system/nls/mycmd
```

If you omit the **%N** variable in a directory specification within the **NLSPATH** environment variable, the application assumes that it defines a catalog name and opens it as such and will not traverse the rest of the search path.

If the **NLSPATH** environment variable is not defined, the **catopen** subroutine uses the default path. See the **/etc/environment** file for the **NLSPATH** default path. If the **LC_MESSAGES** category is set to the default value C, and the **LC__FASTMSG** environment variable is set to `true`, then subsequent calls to the **catgets** subroutine generate pointers to the program-supplied default text.

The **catopen** subroutine treats the first file it finds as a message file. If you specify a non-message file in a **NLSPATH**, for example, **/usr/bin/ls**, **catopen** treats **/usr/bin/ls** as a message catalog. Thus no messages are found and default messages are returned. If you specify **/tmp** in a **NLSPATH**, /**tmp** is opened and searched for messages and default messages are displayed.

Parameters

Item	Description
CatalogName	Specifies the catalog file to open.
Parameter	Determines the environment variable to use in locating the message catalog. If the value of the *Parameter* parameter is 0, use the **LANG** environment variable without regard to the **LC_MESSAGES** category to locate the catalog. If the value of the *Parameter* parameter is the **NL_CAT_LOCALE** macro, use the **LC_MESSAGES** category to locate the catalog.

Return Values

The **catopen** subroutine returns a catalog descriptor. If the **LC_MESSAGES** category is set to the default value C, and the **LC__FASTMSG** environment variable is set to `true`, the **catopen** subroutine returns a value of -1.

If the **LC_MESSAGES** category is not set to the default value C but the **catopen** subroutine returns a value of -1, an error has occurred during creation of the structure of the **nl_catd** data type or the catalog name referred to by the *CatalogName* parameter does not exist.

cbreak, nocbreak, noraw, or raw Subroutine

Purpose

Puts the terminal into or out of CBREAK mode.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int cbreak(void);
```

```
int nocbreak(void);
```

```
int noraw(void);
```

```
int raw(void);
```

Description

The **cbreak** subroutine sets the input mode for the current terminal to cbreak mode and overrides a call to the **raw** subroutine.

The **nocbreak** subroutine sets the input mode for the current terminal to Cooked Mode without changing the state of the **ISIG** and **IXON** flags.

The **noraw** subroutine sets the input mode for the current terminal to Cooked Mode and sets the **ISIG** and **IXON** flags.

The **raw** subroutine sets the input mode for the current terminal to Raw Mode.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

For the **cbreak** and **nocbreak** subroutines:

1. To put the terminal into CBREAK mode, enter:

   ```
   cbreak();
   ```

2. To take the terminal out of CBREAK mode, enter:

   ```
   nocbreak();
   ```

3. To place the terminal into raw mode, use:

   ```
   raw();
   ```

4. To place the terminal out of raw mode, use:

   ```
   noraw();
   ```

For the **noraw** and **raw** subroutines:

1. To place the terminal into raw mode, use:

   ```
   raw();
   ```

2. To place the terminal out of raw mode, use:

   ```
   noraw();
   ```

cbrtf, cbrtl, cbrt, cbrtd32, cbrtd64, and cbrtd128 Subroutines

Purpose

Computes the cube root.

Syntax

```
#include <math.h>

float cbrtf (x)
float x;

long double cbrtl (x)
long double x;

double cbrt (x)
double x;
_Decimal32 cbrtd32 (x)
_Decimal32 x;

_Decimal64 cbrtd64 (x)
_Decimal64 x;
_Decimal128 cbrtd128 (x)
_Decimal128 x;
```

Description

The **cbrtf**, **cbrtl**, **cbrt**, **cbrtd32**, **cbrtd64**, and **cbrtd128** subroutines compute the real cube root of the *x* argument.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **cbrtf**, **cbrtl**, **cbrt**, **cbrtd32**, **cbrtd64**, and **cbrtd128** subroutines return the cube root of *x*.

If *x* is NaN, an NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

ccos, ccosf, or ccosl Subroutine

Purpose

Computes the complex cosine.

Syntax

```
#include <complex.h>

double complex ccos (z)
double complex z;

float complex ccosf (z)
float complex z;
```

```
long double complex ccosl (z)
long double complex z;
```

Description

The **ccos**, **ccosf**, and **ccosl** subroutines compute the complex cosine of *z*.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **ccos**, **ccosf**, and **ccosl** subroutines return the complex cosine value.

ccosh, ccoshf, or ccoshl Subroutine

Purpose

Computes the complex hyperbolic cosine.

Syntax

```
#include <complex.h>

double complex ccosh (z)
double complex z;

float complex ccoshf (z)
float complex z;

long double complex ccoshl (z)
long double complex z;
```

Description

The **ccosh**, **ccoshf**, and **ccoshl** subroutines compute the complex hyperbolic cosine of *z*.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **ccosh**, **ccoshf**, and **ccoshl** subroutines return the complex hyperbolic cosine value.

ccsidtocs or cstoccsid Subroutine

Purpose

Provides conversion between coded character set IDs (CCSID) and code set names.

Library

The iconv Library (**libiconv.a**)

Syntax

```
#include <iconv.h>

CCSID cstoccsid (* Codeset)
const char *Codeset;

char *ccsidtocs ( CCSID)
CCSID CCSID;
```

Description

The **cstoccsid** subroutine returns the CCSID of the code set specified by the *Codeset* parameter. The **ccsidtocs** subroutine returns the code set name of the CCSID specified by *CCSID* parameter. CCSIDs are registered IBM coded character set IDs.

Parameters

Item	Description
Codeset	Specifies the code set name to be converted to its corresponding CCSID.
CCSID	Specifies the CCSID to be converted to its corresponding code set name.

Return Values

If the code set is recognized by the system, the **cstoccsid** subroutine returns the corresponding CCSID. Otherwise, null is returned.

If the CCSID is recognized by the system, the **ccsidtocs** subroutine returns the corresponding code set name. Otherwise, a null pointer is returned.

ceil, ceilf, ceill, ceild32, ceild64, and ceild128 Subroutines

Purpose

Compute the ceiling value.

Syntax

```
#include <math.h>

float ceilf (x)
float x;

long double ceill (x)
long double x;

double ceil (x)
double x;

_Decimal32 ceild32(x)
_Decimal32 x;

_Decimal64 ceild64(x)
_Decimal64 x;

_Decimal128 ceild128(x)
_Decimal128 x;
```

Description

The **ceilf**, **ceill**, **ceil**, **ceild32**, **ceild64**, and **ceild128** subroutines compute the smallest integral value that is not less than *x*.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the smallest integral value to be computed.

Return Values

Upon successful completion, the **ceilf**, **ceill** , **ceil**, **ceild32**, **ceild64**, and **ceild128** subroutines return the smallest integral value that is not less than *x*, expressed as a type **float**, **long double**, **double**, **_Decimal32**, **_Decimal64**, or **_Decimal128** respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

If the correct value would cause overflow, a range error occurs and the **ceilf**, **ceill**, **ceil**, **ceild32**, **ceild64**, and **ceild128** subroutines return the value of the macro **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

cexp, cexpf, or cexpl Subroutine

Purpose

Performs complex exponential computations.

Syntax

```
#include <complex.h>

double complex cexp (z)
double complex z;

float complex cexpf (z)
float complex z;

long double complex cexpl (z)
long double complex z;
```

Description

The **cexp**, **cexpf**, and **cexpl** subroutines compute the complex exponent of *z*, defined as e^z .

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **cexp**, **cexpf**, and **cexpl** subroutines return the complex exponential value of *z*.

cfgetospeed, cfsetospeed, cfgetispeed, or cfsetispeed Subroutine

Purpose

Gets and sets input and output baud rates.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <termios.h>
```

```
speed_t cfgetospeed ( TermiosPointer)
const struct termios *TermiosPointer;
```

```
int cfsetospeed (TermiosPointer,  Speed)
struct termios *TermiosPointer;
speed_t Speed;
```

```
speed_t cfgetispeed (TermiosPointer)
const struct termios *TermiosPointer;
```

```
int cfsetispeed (TermiosPointer, Speed)
struct termios *TermiosPointer;
speed_t Speed;
```

Description

The baud rate subroutines are provided for getting and setting the values of the input and output baud rates in the **termios** structure. The effects on the terminal device described below do not become effective and not all errors are detected until the **tcsetattr** function is successfully called.

The input and output baud rates are stored in the **termios** structure. The supported values for the baud rates are shown in the table that follows this discussion.

The **termios.h** file defines the type **speed_t** as an unsigned integral type.

The **cfgetospeed** subroutine returns the output baud rate stored in the **termios** structure pointed to by the *TermiosPointer* parameter.

The **cfsetospeed** subroutine sets the output baud rate stored in the **termios** structure pointed to by the *TermiosPointer* parameter to the value specified by the *Speed* parameter.

The **cfgetispeed** subroutine returns the input baud rate stored in the **termios** structure pointed to by the *TermiosPointer* parameter.

The **cfsetispeed** subroutine sets the input baud rate stored in the **termios** structure pointed to by the *TermiosPointer* parameter to the value specified by the *Speed* parameter.

Certain values for speeds have special meanings when set in the **termios** structure and passed to the **tcsetattr** function. These values are discussed in the **tcsetattr** subroutine**.**

The following table lists possible baud rates:

Baud Rate Values	
Name	**Description**
B0	Hang up
B5	50 baud
B75	75 baud
B110	110 baud
B134	134 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19200 baud
B38400	38400 baud

The **termios.h** file defines the name symbols of the table.

Parameters

Item	Description
TermiosPointer	Points to a **termios** structure.
Speed	Specifies the baud rate.

Return Values

The **cfgetospeed** and **cfgetispeed** subroutines return exactly the value found in the **termios** data structure, without interpretation.

Both the **cfsetospeed** and **cfsetispeed** subroutines return a value of 0 if successful and -1 if unsuccessful.

Examples

To set the output baud rate to 0 (which forces modem control lines to stop being asserted), enter:

```
cfsetospeed (&my_termios, B0);
tcsetattr (stdout, TCSADRAIN, &my_termios);
```

chacl or fchacl Subroutine

Purpose

Changes the AIXC ACL type access control information of a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/acl.h>
#include <sys/mode.h>
```

```
int chacl ( Path,  ACL,  ACLSize)
char *Path;
struct acl *ACL;
int ACLSize;
```

```
int fchacl ( FileDescriptor, ACL, ACLSize)
int FileDescriptor;
struct acl *ACL;
int ACLSize;
```

Description

The **chacl** and **fchacl** subroutines set the access control attributes of a file according to the AIXC ACL Access Control List (ACL) structure pointed to by the *ACL* parameter. Note that these routines could fail if the current ACL associated with the file system object is of a different type or if the underlying physical file system does not support AIXC ACL type. It is strongly recommended that applications stop using these interfaces and instead make use of **aclx_get** /**aclx_fget** and **aclx_put**/**aclx_fput** subroutines to change the ACL.

Parameters

Item	Description
Path	Specifies the path name of the file.

Item	Description
ACL	Specifies the AIXC ACL to be established on the file. The format of an AIXC ACL is defined in the **sys/acl.h** file and contains the following members: **acl_len** Specifies the size of the ACL (Access Control List) in bytes, including the base entries. **Note:** The entire ACL for a file cannot exceed one memory page (4096 bytes). **acl_mode** Specifies the file mode. The following bits in the **acl_mode** member are defined in the **sys/mode.h** file and are significant for this subroutine: **S_ISUID** Enables the **setuid** attribute on an executable file. **S_ISGID** Enables the **setgid** attribute on an executable file. Enables the group-inheritance attribute on a directory. **S_ISVTX** Enables linking restrictions on a directory. **S_IXACL** Enables extended ACL entry processing. If this attribute is not set, only the base entries (owner, group, and default) are used for access authorization checks. Other bits in the mode, including the following, are ignored: **u_access** Specifies access permissions for the file owner. **g_access** Specifies access permissions for the file group. **o_access** Specifies access permissions for the default class of *others*. **acl_ext[]** Specifies an array of the extended entries for this access control list. The members for the base ACL (owner, group, and others) can contain the following bits, which are defined in the **sys/access.h** file: **R_ACC** Allows read permission. **W_ACC** Allows write permission. **X_ACC** Allows execute or search permission.
FileDescriptor	Specifies the file descriptor of an open file.
ACLSize	Specifies the size of the buffer containing the ACL.

Note: The **chacl** subroutine requires the *Path*, *ACL*, and *ACLSize* parameters. The **fchacl** subroutine requires the *FileDescriptor*, *ACL*, and *ACLSize* parameters.

ACL Data Structure for chacl

Each access control list structure consists of one **struct acl** structure containing one or more **struct acl_entry** structures with one or more **struct** *ace_id* structures.

If the **struct ace_id** structure has *id_type* set to **ACEID_USER** or **ACEID_GROUP**, there is only one *id_data* element. To add multiple IDs to an ACL you must specify multiple **struct ace_id** structures when *id_type* is set to **ACEID_USER** or **ACEID_GROUP**. In this case, no error is returned for the multiple elements, and the access checking examines only the first element. Specifically, the errno value **EINVAL** is not returned for *acl_len* being incorrect in the ACL structure although more than one uid or gid is specified.

Return Values

Upon successful completion, the **chacl** and **fchacl** subroutines return a value of 0. If the **chacl** or **fchacl** subroutine fails, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **chacl** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

Item	Description
ENOTDIR	A component of the *Path* prefix is not a directory.
ENOENT	A component of the *Path* does not exist or has the disallow truncation attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.

The **chacl** or **fchacl** subroutine fails and the access control information for a file remains unchanged if one or more of the following are true:

Item	Description
EROFS	The file specified by the *Path* parameter resides on a read-only file system.
EFAULT	The *ACL* parameter points to a location outside of the allocated address space of the process.
EINVAL	The *ACL* parameter does not point to a valid ACL.
EINVAL	The `acl_len` member in the ACL is not valid.
EIO	An I/O error occurred during the operation.
ENOSPC	The size of the *ACL* parameter exceeds the system limit of one memory page (4KB).
EPERM	The effective user ID does not match the ID of the owner of the file, and the invoker does not have root user authority.

The **fchacl** subroutine fails and the file permissions remain unchanged if the following is true:

Item	Description
EBADF	The file descriptor *FileDescriptor* is not valid.

If Network File System (NFS) is installed on your system, the **chacl** and **fchacl** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

Security

Access Control: The invoker must have search permission for all components of the *Path* prefix.

Auditing Events:

Event	Information
chacl	*Path*
fchacl	*FileDescriptor*

chdir Subroutine

Purpose

Changes the current directory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int chdir ( Path)
const char *Path;
```

Description

The **chdir** subroutine changes the current directory to the directory indicated by the *Path* parameter.

Parameters

Item	Description
Path	A pointer to the path name of the directory. If the *Path* parameter refers to a symbolic link, the **chdir** subroutine sets the current directory to the directory pointed to by the symbolic link. If Network File System (NFS) is installed on the system, this path can cross into another node.

The current directory, also called the current working directory, is the starting point of searches for path names that do not begin with a / (slash). The calling process must have search access to the directory specified by the *Path* parameter.

Return Values

Upon successful completion, the **chdir** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to identify the error.

Error Codes

The **chdir** subroutine fails and the current directory remains unchanged if one or more of the following are true:

Item	Description
EACCES	Search access is denied for the named directory.
ENOENT	The named directory does not exist.
ENOTDIR	The path name is not a directory.

The **chdir** subroutine can also be unsuccessful for other reasons. See *Base Operating System error codes for services* that require path-name resolution for a list of additional error codes.

If NFS is installed on the system, the **chdir** subroutine can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

checkauths Subroutine

Purpose

Compares the passed-in list of authorizations to the authorizations associated with the current process.

Library

Security Library (**libc.a**)

Syntax

```
# include <usersec.h>

int checkauths(CommaListOfAuths, Flag)
    char *CommaListOfAuths;
    int Flag;
```

Description

The **checkauths** subroutine compares a comma-separated list of authorizations specified in the *CommaListOfAuths* parameter with the authorizations associated with the calling process. The *Flag* parameter specifies the type of checks the subroutine performs. If the *Flag* parameter specifies the **CHECK_ANY** value, and the calling process contains any of the authorizations specified in the *CommaListOfAuths* parameter, the subroutine returns the value of zero. If the *Flag* parameter specifies the **CHECK_ALL** value, and the calling process contains all of the authorizations that are specified in the *CommaListOfAuths* parameter, the subroutine returns the value of zero.

You can use the **checkauths** subroutine for both Enhanced and Legacy RBAC modes. The set of authorizations that are available to a process depends on the mode that the system is operating in. In Enhanced RBAC Mode, the set of authorizations comes from the current active role set of the process, while in Legacy RBAC Mode, the set of authorizations comes from all of the roles associated with the process owner.

Parameters

Item	Description
CommaListOfAuths	Specifies one or more authorizations. The authorizations are separated by commas.
Flag	Specifies an integer value that controls the type of checking for the subroutine to perform. The *Flag* parameter contains the following possible values: **CHECK_ANY** Returns **0** if the process has any of the authorizations that the *CommaListOfAuths* parameter specifies. **CHECK_ALL** Returns **0** if the process has all of the authorizations that the *CommaListOfAuths* parameter specifies.

Return Values

If the process matches the required set of authorizations, the **checkauths** subroutine returns the value of zero. Otherwise, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **checkauths** subroutine returns -1, one of the following **errno** values can be set:

Item	Description
EINVAL	The *CommaListOfAuths* parameter is NULL or the NULL string.
EINVAL	The *Flag* parameter contains an unrecognized flag.
ENOMEM	Memory cannot be allocated.

chmod, fchmod, or fchmodat Subroutine

Purpose

Changes file system object base file mode bits.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/stat.h>
```

```
int chmod (Path,  Mode)
const char *Path;
mode_t Mode;
```

```
int fchmod (FileDescriptor, Mode)
int FileDescriptor;
mode_t Mode;
```

```
int fchmodat (DirFileDescriptor, Path, Mode, Flag)
int DirFileDescriptor;
const char *Path;
mode_t Mode;
int  Flag;
```

Description

The **chmod** subroutine sets the access permissions of the file specified by the *Path* parameter. If Network File System (NFS) is installed on your system, this path can cross into another node.

Use the **fchmod** subroutine to set the access permissions of an open file pointed to by the *FileDescriptor* parameter.

If *FileDescriptor* references a shared memory object, the **fchmod** subroutine affects the **S_IRUSR**, **S_IWUSR**, **S_IRGRP**, **S_IWGRP**, **S_IROTH**, and **S_IWOTH** file permission bits.

The access control information is set according to the *Mode* parameter. Note that these routines will replace any existing ACL associated with the file system object.

The **fchmodat** subroutine is equivalent to the **chmod** subroutine if the *Path* parameter specifies an absolute path or if the *DirFileDescriptor* parameter is set to **AT_FDCWD**. The file to be changed is determined by the relative path to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory. If the directory is opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted by using the current permissions of the directory. If the directory is opened with the **O_SEARCH** open flag, the subroutine does not perform the check.

Parameters

Item	Description
FileDescriptor	Specifies the file descriptor of an open file or shared memory object.

Item	Description
Mode	Specifies the bit pattern that determines the access permissions. The *Mode* parameter is constructed by logically ORing one or more of the following values, which are defined in the **sys/mode.h** file: **S_ISUID** Enables the **setuid** attribute for an executable file. A process executing this program acquires the access rights of the owner of the file. **S_ISGID** Enables the **setgid** attribute for an executable file. A process executing this program acquires the access rights of the group of the file. Also, enables the group-inheritance attribute for a directory. Files created in this directory have a group equal to the group of the directory. The following attributes apply only to files that are directly executable. They have no meaning when applied to executable text files such as shell scripts and **awk** scripts. **S_ISVTX** Enables the **link/unlink** attribute for a directory. Files cannot be linked to in this directory. Files can only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the directory. **S_ISVTX** Enables the **save text** attribute for an executable file. The program is not unmapped after usage. This attribute can only be enabled by the root user. When specified by anyone else, this attribute is ignored. **S_ENFMT** Enables enforcement-mode record locking for a regular file. File locks requested with the **lockf** subroutine are enforced. **S_IRUSR** Permits the file's owner to read it. **S_IWUSR** Permits the file's owner to write to it. **S_IXUSR** Permits the file's owner to execute it (or to search the directory). **S_IRGRP** Permits the file's group to read it. **S_IWGRP** Permits the file's group to write to it. **S_IXGRP** Permits the file's group to execute it (or to search the directory). **S_IROTH** Permits others to read the file. **S_IWOTH** Permits others to write to the file. **S_IXOTH** Permits others to execute the file (or to search the directory). Other mode values exist that can be set with the **mknod** subroutine but not with the **chmod** subroutine.
Path	Specifies the path name of the file. For **fchmodat**, if the *DirFileDescriptor* is specified and Path is relative, then the *DirFileDescriptor* specifies the effective current working directory for the *Path*.

Item	Description
DirFileDescriptor	Specifies the file descriptor of an open directory, which is used as the effective current working directory for the *Path* parameter. If *DirFileDescriptor* equals **AT_FDCWD**, the *DirFileDescriptor* parameter is ignored and the *Path* argument specifies the complete file.
Flag	Specifies a bit field argument. If the *Flag* parameter contains the **AT_SYMLINK_NOFOLLOW** bit and the *Path* parameter specifies a symbolic link, the mode of the symbolic link is changed.

Return Values

Upon successful completion, the **chmod**, **fchmod**, and **fchmodat** subroutines return a value of 0. If the **chmod**, **fchmod**, or **fchmodat** subroutine is unsuccessful, a value of -1 is returned, and the **errno** global variable is set to identify the error.

Error Codes

The **chmod** or **fchmodat** subroutine is unsuccessful and the file permissions remain unchanged if one of the following is true:

Item	Description
ENOTDIR	A component of the *Path* prefix is not a directory.
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENOENT	The named file does not exist.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.

The **fchmod** subroutine is unsuccessful and the file permissions remain unchanged if the following is true:

Item	Description
EBADF	The value of the *FileDescriptor* parameter is not valid.

The **chmod**, **fchmod** or **fchmodat** subroutine is unsuccessful and the access control information for a file remains unchanged if one of the following is true:

Item	Description
EPERM	The effective user ID does not match the owner of the file, and the process does not have appropriate privileges.
EROFS	The named file resides on a read-only file system.
EIO	An I/O error occurred during the operation.

If NFS is installed on your system, the **chmod** and **fchmod** subroutines can also be unsuccessful if the following is true:

Item	Description
ESTALE	The root or current directory of the process is located in a virtual file system that has been unmounted.
ETIMEDOUT	The connection timed out.

The **fchmodat** subroutine is unsuccessful and the file permissions remain unchanged if one of the following is true:

Item	Description
EBADF	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* argument is neither **AT_FDCWD** nor a valid file descriptor.
EINVAL	The value of the *Flag* argument is not valid.
ENOTDIR	The *Path* parameter is not an absolute path and *DirFileDescriptor* is a file descriptor but is not associated with a directory.

Security

Access Control: The invoker must have search permission for all components of the *Path* prefix.

If you receive the **EBUSY** error, toggle the **enforced locking** attribute in the *Mode* parameter and retry your operation. The **enforced locking** attribute should never be used on a file that is part of the Trusted Computing Base.

chown, fchown, lchown, chownx, fchownx, chownxat, or fchownat Subroutine

Purpose

Changes file ownership.

Library

Standard C Library (**libc.a**)

Syntax

#include <sys/types.h> #include <unistd.h>

int chown (*Path*, *Owner*, *Group* **) const char ***Path**; uid_t** *Owner***; gid_t** *Group***;**

int fchown (*FileDescriptor*, *Owner*, Group **)**

int *FileDescriptor***; uid_t** *Owner***; gid_t** *Group***;**

int lchown (*Path*, *Owner*, *Group* **)**

const char **fname* **uid_t** *uid* **gid_t***gid*

#include <sys/types.h>

#include <sys/chownx.h>

int chownx (*Path*, *Owner*, *Group*, *Flags* **)**

char ***Path; uid_t** *Owner***; gid_t** *Group***; int** *Flags***;**

int fchownx (*FileDescriptor*, *Owner*, *Group*, *Flags* **)**

int *FileDescriptor*; **uid_t** *Owner*; **gid_t** *Group*; **int** *Flags*;

int chownxat (*DirFileDescriptor*, *Path*, *Owner*, Group, *Flags*)

int *DirFileDescriptor*;

char * *Path*;

uid_t *Owner*;

gid_t *Group*;

int *Flags*;

int fchownat (*DirFileDescriptor*, *Path*, *Owner*, Group, *Flag*)

int *DirFileDescriptor*;

char*Path*;

uid_t *Owner*;

gid_t *Group*;

int *Flag;*

Description

The **chown**, **chownx**, **fchown**, **fchownx**, **chownxat**, **fchownat**, and **lchown** subroutines set the file owner and group IDs of the specified file system object. Root user authority is required to change the owner of a file.

A function **lchown** function sets the owner ID and group ID of the named file similarity to **chown** function except in the case where the named file is a symbolic link. In this case **lchown** function changes the ownership of the symbolic link file itself, while **chown** function changes the ownership of the file or directory to which the symbolic link refers.

The **chownxat** subroutine is equivalent to the **chownx** subroutine and the **fchownat** subroutine is equivalent to the **chown** or the **lchown** subroutine if the *Path* parameter specifies an absolute path or if the *DirFileDescriptor* parameter is set to **AT_FDCWD**. The file to be changed is determined by the relative path to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory. If the directory is opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted by using the current permissions of the directory. If the directory is opened with the **O_SEARCH** open flag, the subroutine does not perform the check.

Parameters

Item	Description
FileDescriptor	Specifies the file descriptor of an open file.
Flags	Specifies whether the file owner ID or group ID should be changed. This parameter is constructed by logically ORing the following values: **T_OWNER_AS_IS** Ignores the value specified by the Owner parameter and leaves the owner ID of the file unaltered. **T_GROUP_AS_IS** Ignores the value specified by the Group parameter and leaves the group ID of the file unaltered.
Flag	Specifies a bit field. If the AT_SYMLINK_NOFOLLOW bit is set and the Path specifies a symbolic link, then the owner and group of the symbolic link is changed.

Item	Description
Group	Specifies the new group of the file. For the **chown**, **fchown**, **fchownat**, and **lchown** commands, if this value is -1, the group is not changed. (A value of -1 indicates only that the group is not changed; it does not indicate a group that is not valid. An owner or group ID cannot be invalid.) For the **chownx**, **chownxat**, and **fchownx** commands, the subroutines change the Group to -1 if -1 is supplied for Group and **T_GROUP_AS_IS** is not set.
Owner	Specifies the new owner of the file. For the **chown**, **fchown**, **fchownat**, and **lchown** commands, if this value is -1, the group is not changed. (A value of -1 indicates only that the group is not changed; it does not indicate a group that is not valid. An owner or group ID cannot be invalid.) For the **chownx**, **chownxat**, and **fchownx** commands, the subroutines change the Owner to -1 if -1 is supplied for Owner and **T_OWNER_AS_IS** is not set.
Path	Specifies the path name of the file. For **chownxat** and **fchownat**, if the *DirFileDescriptor* is specified and Path is relative, then the *DirFileDescriptor* specifies the effective current working directory for the Path.
DirFileDescriptor	Specifies the file descriptor of an open directory, which is used as the effective current working directory for the *Path* parameter. If the *DirFileDescriptor* parameter equals **AT_FDCWD**, the *DirFileDescriptor* parameter is ignored and the *Path* argument specifies the complete file.

Return Values

Upon successful completion, the **chown**, **chownx**, **fchown**, **fchownx**, **chownxat**, **fchownat**, and **lchown** subroutines return a value of 0. If the **chown**, **chownx**, **fchown**, **fchownx**, **chownxat**, **fchownat**, or **lchown** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **chown**, **fchownat**, **chownx**, **chownxat**, or **lchown** subroutine is unsuccessful and the owner and group of a file remain unchanged if one of the following is true:

Item	Description
EACCES	Search permission is denied on a component of the *Path* parameter.
EDQUOT	The new group for the file system object cannot be set because the group's quota of disk blocks or i-nodes has been exhausted on the file system.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.
EINVAL	The owner or group ID supplied is not valid.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.
ENOENT	A symbolic link was named, but the file to which it refers does not exist; or a component of the *Path* parameter does not exist; or the process has the **disallow truncation** attribute set; or the *Path* parameter is null.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The effective user ID does not match the owner of the file, and the calling process does not have the appropriate privileges.

Item	Description
EROFS	The named file resides on a read-only file system.
ESTALE	The root or current directory of the process is located in a virtual file system that has been unmounted.

The **fchown** or **fchownx** subroutine is unsuccessful and the file owner and group remain unchanged if one of the following is true:

Item	Description
EBADF	The named file resides on a read-only file system.
EDQUOT	The new group for the file system object cannot be set because the group's quota of disk blocks or i-nodes has been exhausted on the file system.
EIO	An I/O error occurred during the operation.

The **chownxat** or the **fchownat** subroutine is unsuccessful and the file owner and group remain unchanged if one of the following is true:

Item	Description
EBADF	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* argument is neither **AT_FDCWD** nor a valid file descriptor.
EINVAL	The value of the *Flag* parameter is not valid.
ENOTDIR	The *Path* parameter is not an absolute path and *DirFileDescriptor* is a file descriptor but is not associated with a directory.

Security

Access Control: The invoker must have search permission for all components of the *Path* parameter.

chpass Subroutine

Purpose

Changes user passwords.

Library

Standard C Library (**libc.a**)

Thread Safe Security Library (**libs_r.a**)

Syntax

```
int chpass (UserName, Response, Reenter, Message)
char *UserName;
char *Response;
int *Reenter;
char **Message;
```

Description

The **chpass** subroutine maintains the requirements that the user must meet to change a password. This subroutine is the basic building block for changing passwords and handles password changes for local, NIS, and DCE user passwords.

The *Message* parameter provides a series of messages asking for old and new passwords, or providing informational messages, such as the reason for a password change failing. The first *Message* prompt is a prompt for the old password. This parameter does not prompt for the old password if the user has a real user ID of 0 (zero) and is changing a local user, or if the user has no current password. The **chpass** subroutine does not prompt a user with root authority for an old password. It informs the program that no message was sent and that it should invoke **chpass** again. If the user satisfies the first *Message* parameter's prompt, the system prompts the user to enter the new password. Each message is contained in the *Message* parameter and is displayed to the user. The *Response* parameter returns the user's response to the **chpass** subroutine.

The *Reenter* parameter indicates when a user has satisfied all prompt messages. The parameter remains nonzero until a user has passed all prompts. After the returned value of *Reenter* is 0, the return code signals whether the password change has succeeded or failed. When progressing through prompts for a user, the value of *Reenter* must be maintained by the caller between invocations of **chpass**.

The **chpass** subroutine maintains internal state information concerning the next prompt message to present to the user. If the calling program supplies a different user name before all prompt messages are complete for the user, the internal state information is reset and prompt messages begin again. State information is also kept in the *Reenter* variable. The calling program must maintain the value of *Reenter* between calls to **chpass**.

The **chpass** subroutine determines the administration domain to use during password changes. It determines if the user is defined locally, defined in Network Information Service (NIS), or defined in Distributed Computing Environment (DCE). Password changes occur only in these domains. System administrators may override this convention with the registry value in the **/etc/security/user** file. If the registry value is defined, the password change can only occur in the specified domain. System administrators can use this registry value if the user is administered on a remote machine that periodically goes down. If the user is allowed to log in through some other authentication method while the server is down, password changes remain to follow only the primary server.

The **chpass** subroutine allows the user to change passwords in two ways. For normal (non-administrative) password changes, the user must supply the old password, either on the first call to the **chpass** subroutine or in response to the first message from **chpass**. If the user is root, real user ID of 0, local administrative password changes are handled by supplying a null pointer for the *Response* parameter during the initial call

Users that are not administered locally are always queried for their old password.

The **chpass** subroutine is always in one of the following states:

1. Initial state: The caller invokes the **chpass** subroutine with NULL *response* parameter and receives the initial password prompt in the *message* parameter.
2. Verify initial password: The caller invokes the **chpass** subroutine with the results of prompting the user with earlier *message* parameter as the *response* parameter. The caller is given a prompt to enter the new password in the *message* parameter.
3. Enter new password: The caller invokes the **chpass** subroutine with the results of prompting user with the new password prompt in the *response* parameter. The caller will be given a prompt to repeat the new password in the *message* parameter.
4. Verify new password: The caller invokes the **chpass** subroutine with the results of prompting the user to repeat the new password in the *response* parameter. The **chpass** subroutine then performs the actual password change.

Any step in the above process can result in the **chpass** subroutine terminating the dialog. This is signalled when the *reenter* variable is set to 0. The return code indicates the nature of the failure.

Note: Set the setuid and owner to root for your own programs that use the **chpass** subroutine.

Parameters

Item	Description
UserName	Specifies the user's name whose password is to be changed.
Response	Specifies a character string containing the user's response to the last prompt.
Reenter	Points to a Boolean value used to signal whether the **chpass** subroutine has completed processing. If the *Reenter* parameter is a nonzero value, the **chpass** subroutine expects the user to satisfy the prompt message provided by the *Message* parameter. If the *Reenter* parameter is 0, the **chpass** subroutine has completed processing.
Message	Points to a pointer that the **chpass** subroutine allocates memory for and fills in. This replacement string is then suitable for printing and issues challenge messages (if the *Reenter* parameter is a nonzero value). The string can also issue informational messages such as why the user failed to change the password (if the *Reenter* parameter is 0). The calling application is responsible for freeing this memory.

Return Values

Upon successful completion, the **chpass** subroutine returns a value of 0. If the **chpass** subroutine is unsuccessful, it returns the following values:

Item	Description
-1	Indicates the call failed in the thread safe library **libs_r.a**. **ERRNO** will indicate the failure code.
1	Indicates that the password change was unsuccessful and the user should attempt again. This return value occurs if a password restriction is not met, such as if the password is not long enough.
2	Indicates that the password change was unsuccessful and the user should not attempt again. This return value occurs if the user enters an incorrect old password or if the network is down (the password change cannot occur).

Error Codes

The **chpass** subroutine is unsuccessful if one of the following values is true:

Item	Description
ENOENT	Indicates that the user cannot be found.
ESAD	Indicates that the user did not meet the criteria to change the password.
EPERM	Indicates that the user did not have permission to change the password.
EINVAL	Indicates that the parameters are not valid.
ENOMEM	Indicates that memory allocation (malloc) failed.

chpassx Subroutine

Purpose

Changes multiple method passwords.

Library

Standard C Library (**libc.a**)

Syntax

```
int chpassx (UserName, Response, Reenter, Message, State)
char *UserName;
char *Response;
int *Reenter;
char **Message;
void **State;
```

Description

The **chpassx** subroutine maintains the requirements that the user must meet to change a password. This subroutine is the basic building block for changing passwords, and it handles password changes for local, NIS, and loadable authentication module user passwords. It uses information provided by the **authenticatex** and **passwdexpiredx** subroutines to indicate which passwords were used when a user authenticated and whether or not those passwords are expired.

The *Message* parameter provides a series of messages asking for old and new passwords, or providing informational messages, such as the reason for a password change failing. The first *Message* prompt is a prompt for the old password. This parameter does not prompt for the old password if the user has a real user ID of 0 and is changing a local user, or if the user has no current password. The **chpassx** subroutine does not prompt a user with root authority for an old password when only a local password is being changed. It informs the program that no message was sent and that it should invoke **chpass** again. If the user satisfies the first *Message* parameter's prompt, the system prompts the user to enter the new password. Each message is contained in the *Message* parameter and is displayed to the user. The *Response* parameter returns the user's response to the **chpass** subroutine.

The *Reenter* parameter remains a nonzero value until the user satisfies all of the prompt messages or until the user incorrectly responds to a prompt message. When the *Reenter* parameter is 0, the return code signals whether the password change completed or failed. The calling application must initialize the *Reenter* parameter to 0 before the first call to the **chpassx** subroutine and the application cannot modify the *Reenter* parameter until the sequence of **chpassx** subroutine calls has completed.

The **authenticatex** subroutine ascertains the authentication domains the user can attempt. The subroutine uses the **SYSTEM** attribute for the user. Each token that is displayed in the **SYSTEM** line corresponds to a method that can be dynamically loaded and processed. Likewise, the system can provide multiple or alternate authentication paths.

The *State* parameter contains information from an earlier call to the **authenticatex** or **passwdexpirex** subroutines. That information indicates which administration domains were used when the user was authenticated and which passwords have expired and can be changed by the user. The *State* parameter must be initialized to null when the **chpassx** subroutine is not being called after an earlier call to the **authenticatex** or **passwdexpiredx** subroutines, or if the calling program does not wish to use the information from an earlier call.

The **chpassx** subroutine maintains internal state information concerning the next prompt message to present to the user. If the calling program supplies a different user name before all prompt messages are complete for the user, the internal state information is reset and prompt messages begin again.

The **chpassx** subroutine determines the administration domain to use during password changes. It determines if the user is defined locally, defined in Network Information Service (NIS), defined in Distributed Computing Environment (DCE), or defined in another administrative domain supported by a loadable authentication module. Password changes use the user's **SYSTEM** attribute and information in the *State* parameter. When the *State* parameter includes information from an earlier call to the **authenticatex** subroutine, only the administrative domains that were used for authentication are changed. When the *State* parameter includes information from an earlier call to the **passwdexpiredx** subroutine, only the administrative domains that have expired passwords are changed. The *State* parameter can contain information from calls to both **authenticatex** and **passwdexpiredx**, in which case

passwords that were used for authentication are changed, even if they are not expired, so that passwords remain synchronized between administrative domains.

The **chpassx** subroutine allows the user to change passwords in two ways. For normal (nonadministrative) password changes, the user must supply the old password, either on the first call to the **chpassx** subroutine or in response to the first message from **chpassx**. If the user is root (with a real user ID of 0), local administrative password changes are handled by supplying a null pointer for the *Response* parameter during the initial call.

Users that are not administered locally are always queried for their old password.

The **chpassx** subroutine is always in one of three states: entering the old password, entering the new password, or entering the new password again. If any of these states do not need to be complied with, the **chpassx** subroutine returns a null challenge.

Parameters

Item	Description
Message	Points to a pointer that the **chpassx** subroutine allocates memory for and fills in. This replacement string is then suitable for printing and issues challenge messages (if the *Reenter* parameter is a nonzero value). The string can also issue informational messages, such as why the user failed to change the password (if the *Reenter* parameter is 0). The calling application is responsible for freeing this memory.
Reenter	Points to an integer value used to signal whether the **chpassx** subroutine has completed processing. If the *Reenter* parameter is a nonzero value, the **chpassx** subroutine expects the user to satisfy the prompt message provided by the *Message* parameter. If the Reenter parameter is 0, the **chpassx** subroutine has completed processing.
Response	Specifies a character string containing the user's response to the last prompt.
State	Points to a pointer that the **chpassx** subroutine allocates memory for and fills in. The *State* parameter can also be the result of an earlier call to the **authenticatex** or **passwdexpiredx** subroutines. This parameter contains information about each password that has been changed for the user. The calling application is responsible for freeing this memory after the **chpassx** subroutine has completed.
UserName	Specifies the user's name whose password is to be changed.

Return Values

Upon successful completion, the **chpassx** subroutine returns a value of 0. If this subroutine fails, it returns the following values:

Item	Description
-1	The call failed in the **libs_r.a** thread safe library. **errno** indicates the failure code.
1	The password change was unsuccessful and the user should try again. This return value occurs if a password restriction is not met (for example, the password is not long enough).
2	The password change was unsuccessful and the user should not try again. This return value occurs if the user enters an incorrect old password or if the network is down (the password change cannot occur).

Error Codes

The **chpassx** subroutine is unsuccessful if one of the following values is true:

Item	Description
EINVAL	The parameters are not valid.
ENOENT	The user cannot be found.
ENOMEM	Memory allocation (malloc) failed.
EPERM	The user did not have permission to change the password.
ESAD	The user did not meet the criteria to change the password.

chprojattr Subroutine

Purpose

Updates and modifies the project attributes in kernel project registry for the given project.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>
chprojattr(struct project *, int cmd)
```

Description

The **chprojattr** subroutine alters the attributes of a project defined in the kernel project registry. A pointer to struct project containing the project definition and the operation command is sent as input arguments. The following operations are permitted:

- PROJ_ENABLE_AGGR - Enables aggregation for the specified project
- PROJ_DISABLE_AGGR - Disables aggregation for the specified project

If PROJ_ENABLE_AGGR is passed, then the aggregation status bit is set to 1. If PROJ_DISABLE_AGGR is passed, then the aggregation status bit set to 0.

Note: To initialize the project structure, the user must call the **getprojdef** subroutine before calling the **chprojattr** subroutine.

Parameters

Item	Description
project	Pointer containing the project definition.
cmd	An integer command indicating whether to perform a set or clear operation.

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success

Item	Description
-1	Failure

Error Codes

Item	Description
EINVAL	Invalid arguments passed. The passed command flag is invalid or the passed pointer is NULL.
ENONENT	Project not found.

chprojattrdb Subroutine

Purpose

Updates the project attributes in the project database.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

chprojattrdb(void *handle, struct project *project, int cmd)
```

Description

The **chprojattrdb** subroutine alters the attributes of the named project in the specified project database, which is controlled through the *handle* parameter. The following commands are permitted:

- **PROJ_ENABLE_AGGR** — Enables aggregation for the specified project
- **PROJ_DISABLE_AGGR** — Disables aggregation for the specified project

The project database must be initialized before calling this subroutine. The **projdballoc** subroutine is provided for this purpose. The **chprojattrdb** subroutine must be called after the **getprojdb** subroutine, which sets the record pointer to point to the project that needs to be modified.

Note: The **chprojattrdb** subroutine must be called after the **getprojdb** subroutine, which makes the named project the current project.

Parameters

Item	Description
handle	Pointer to the handle allocated for the project database.
project	Pointer containing the project definition.
cmd	An integer command indicating whether to perform a set or clear operation.

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Invalid arguments passed. The passed command flag is invalid or the passed pointer is NULL.
ENONENT	Project not found.

chroot Subroutine

Purpose

Changes the effective root directory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int chroot (const char * Path)
char *Path;
```

Description

The **chroot** subroutine causes the directory named by the *Path* parameter to become the effective root directory. If the *Path* parameter refers to a symbolic link, the **chroot** subroutine sets the effective root directory to the directory pointed to by the symbolic link. If Network File System (NFS) is installed on your system, this path can cross into another node.

The effective root directory is the starting point when searching for a file's path name that begins with / (slash). The current directory is not affected by the **chroot** subroutine.

The calling process must have root user authority in order to change the effective root directory. The calling process must also have search access to the new effective root directory.

The .. (double period) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, this directory cannot be used to access files outside the subtree rooted at the effective root directory.

Parameters

Item	Description
Path	Pointer to the new effective root directory.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **chroot** subroutine fails and the effective root directory remains unchanged if one or more of the following are true:

Item	Description
ENOENT	The named directory does not exist.
EACCES	The named directory denies search access.
EPERM	The process does not have root user authority.

The **chroot** subroutine can be unsuccessful for other reasons. See *Appendix A. Base Operating System Error Codes for Services* that Require Path-Name Resolution for a list of additional errors.

If NFS is installed on the system, the **chroot** subroutine can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

chssys Subroutine

Purpose

Modifies the subsystem objects associated with the *SubsystemName* parameter.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
```

```
int chssys( SubsystemName,  SRCSubsystem)
char *SubsystemName;
struct SRCsubsys *SRCSubsystem;
```

Description

The **chssys** subroutine modifies the subsystem objects associated with the specified subsystem with the values in the **SRCsubsys** structure. This action modifies the objects associated with subsystem in the following object classes:

- Subsystem Environment
- Subserver Type
- Notify

The Subserver Type and Notify object classes are updated only if the subsystem name has been changed.

The **SRCsubsys** structure is defined in the **/usr/include/sys/srcobj.h** file.

The program running with this subroutine must be running with the group system.

Parameters

Item	Description
SRCSubsystem	Points to the **SRCsubsys** structure.
SubsystemName	Specifies the name of the subsystem.

Return Values

Upon successful completion, the **chssys** subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **odmerrno** variable is set to indicate the error, or a System Resource Controller (SRC) error code is returned.

Error Codes

The **chssys** subroutine is unsuccessful if one or more of the following are true:

Item	Description
SRC_NONAME	No subsystem name is specified.
SRC_NOPATH	No subsystem path is specified.
SRC_BADNSIG	Invalid stop normal signal.
SRC_BADFSIG	Invalid stop force signal.
SRC_NOCONTACT	Contact not signal, sockets, or message queues.
SRC_SSME	Subsystem name does not exist.
SRC_SUBEXIST	New subsystem name is already on file.
SRC_SYNEXIST	New subsystem synonym name is already on file.
SRC_NOREC	The specified **SRCsubsys** record does not exist.
SRC_SUBSYS2BIG	Subsystem name is too long.
SRC_SYN2BIG	Synonym name is too long.
SRC_CMDARG2BIG	Command arguments are too long.
SRC_PATH2BIG	Subsystem path is too long.
SRC_STDIN2BIG	stdin path is too long.
SRC_STDOUT2BIG	stdout path is too long.
SRC_STDERR2BIG	stderr path is too long.
SRC_GRPNAM2BIG	Group name is too long.

Security

Privilege Control: This command has the Trusted Path attribute. It has the following kernel privilege:

```
SET_PROC_AUDIT kernel privilege
```

Item	Description
Files Accessed:	

Mode	File
644	**/etc/objrepos/SRCsubsys**

Mode	File
644	**/etc/objrepos/SRCsubsvr**
644	**/etc/objrepos/SRCnotify**

Auditing Events:

Event	Information
SRC_Chssys	

Files

Item	Description
/etc/objrepos/SRCsubsys	SRC Subsystem Configuration object class.
/etc/objrepos/SRCsubsvr	SRC Subserver Configuration object class.
/etc/objrepos/SRCnotify	SRC Notify Method object class.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

cimag, cimagf, or cimagl Subroutine

Purpose

Performs complex imaginary computations.

Syntax

```
#include <complex.h>

double cimag (z)
double complex z;

float cimagf (z)
float complex z;

long double cimagl (z)
long double complex z;
```

Description

The **cimag**, **cimagf**, and **cimagl** subroutines compute the imaginary part of *z*.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **cimag**, **cimagf**, and **cimagl** subroutines return the imaginary part value (as a real).

ckuseracct Subroutine

Purpose

Checks the validity of a user account.

Library

Security Library (**libc.a**)

Syntax

```
#include <login.h>

int ckuseracct ( Name,  Mode,  TTY)
char *Name;
int Mode;
char *TTY;
```

Description

Note: This subroutine is obsolete and is provided only for backwards compatibility. Use the **loginrestrictions** subroutine, which performs a superset of the functions of the **ckuseracct** subroutine, instead.

The **ckuseracct** subroutine checks the validity of the user account specified by the *Name* parameter. The *Mode* parameter gives the mode of the account usage, and the *TTY* parameter defines the terminal being used for the access. The **ckuseracct** subroutine checks for the following conditions:

- Account existence
- Account expiration

The *Mode* parameter specifies other mode-specific checks.

Parameters

Item	Description
Name	Specifies the login name of the user whose account is to be validated.
Mode	Specifies the manner of usage. Valid values as defined in the **login.h** file are listed below. The *Mode* parameter must be one of these or 0: **S_LOGIN** Verifies that local logins are permitted for this account. **S_SU** Verifies that the **su** command is permitted and that the current process has a group ID that can invoke the **su** command to switch to the account. **S_DAEMON** Verifies the account can be used to invoke daemon or batch programs using the **src** or **cron** subsystems. **S_RLOGIN** Verifies the account can be used for remote logins using the **rlogind** or **telnetd** programs.
TTY	Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no TTY origin checking is done.

Security

Item	Description

Files Accessed:

Mode	File
r	**/etc/passwd**
r	**/etc/security/user**

Return Values

If the account is valid for the specified usage, the **ckuseracct** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to the appropriate error code.

Error Codes

The **ckuseracct** subroutine fails if one or more of the following are true:

Item	Description
ENOENT	The user specified in the *Name* parameter does not have an account.
ESTALE	The user's account is expired.
EACCES	The specified terminal does not have access to the specified account.
EACCES	The *Mode* parameter is **S_SU**, and the current process is not permitted to use the **su** command to access the specified user.
EACCES	Access to the account is not permitted in the specified *Mode*.
EINVAL	The *Mode* parameter is not one of **S_LOGIN**, **S_SU**, **S_DAEMON**, **S_RLOGIN**.

ckuserID Subroutine

Purpose

Authenticates the user.

Note: This subroutine is obsolete and is provided for backwards compatibility. Use the **authenticate** subroutine, instead.

Library

Security Library (**libc.a**)

Syntax

```
#include <login.h>
int ckuserID ( User,  Mode)
int Mode;
char *User;
```

Description

The **ckuserID** subroutine authenticates the account specified by the *User* parameter. The mode of the authentication is given by the *Mode* parameter. The **login** and **su** commands continue to use the **ckuserID** subroutine to process the **/etc/security/user auth1** and **auth2** authentication methods.

The **ckuserID** subroutine depends on the **authenticate** subroutine to process the **SYSTEM** attribute in the **/etc/security/user** file. If authentication is successful, the **passwdexpired** subroutine is called.

Errors caused by grammar or load modules during a call to the **authenticate** subroutine are displayed to the user if the user was authenticated. These errors are audited with the **USER_Login** audit event if the user failed authentication.

Parameters

Item	Description
User	Specifies the name of the user to be authenticated.
Mode	Specifies the mode of authentication. This parameter is a bit mask and may contain one or more of the following values, which are defined in the **login.h** file: **S_PRIMARY** The primary authentication methods defined for the *User* parameter are checked. All primary authentication checks must be passed. **S_SECONDARY** The secondary authentication methods defined for the *User* parameter are checked. Secondary authentication checks are not required to be successful. Primary and secondary authentication methods for each user are set in the /**etc**/**security**/**user** file by defining the **auth1** and **auth2** attributes. If no primary methods are defined for a user, the **SYSTEM** attribute is assumed. If no secondary methods are defined, there is no default.

Security

Item	Description

Files Accessed:

Mode	File
r	/**etc**/**passwd**
r	/**etc**/**security**/**passwd**
r	/**etc**/**security**/**user**
r	/**etc**/**security**/**login.cfg**

Return Values

If the account is valid for the specified usage, the **ckuserID** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **ckuserID** subroutine fails if one or more of the following are true:

Item	Description
ESAD	Security authentication failed for the user.
EINVAL	The *Mode* parameter is neither **S_PRIMARY** nor **S_SECONDARY** or the *Mode* parameter is both **S_PRIMARY** and **S_SECONDARY**.

class, _class, finite, isnan, or unordered Subroutines

Purpose

Determines classifications of floating-point numbers.

Libraries

IEEE Math Library (**libm.a**) or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
#include <float.h>
```

```
int
class( x)
double x;
```

```
#include <math.h>
#include <float.h>
```

```
int
_class( x)
double x;
```

```
#include <math.h>
```

```
int finite(x)
double x;
```

```
#include <math.h>
```

```
int isnan(x)
double x;
```

```
#include <math.h>
```

```
int unordered(x, y)
double x, y;
```

Description

The **class** subroutine, **_class** subroutine, **finite** subroutine, **isnan** subroutine, and **unordered** subroutine determine the classification of their floating-point value. The **unordered** subroutine determines if a floating-point comparison involving *x* and *y* would generate the IEEE floating-point unordered condition (such as whether *x* or *y* is a NaN).

The **class** subroutine returns an integer that represents the classification of the floating-point *x* parameter. Since **class** is a reversed key word in C++. The **class** subroutine can not be invoked in a C++ program. The **_class** subroutine is an interface for C++ program using the **class** subroutine. The interface and the return value for class and **_class** subroutines are identical. The values returned by the **class** subroutine are defined in the **float.h** header file. The return values are the following:

Item	Description
FP_PLUS_NORM	Positive normalized, nonzero *x*
FP_MINUS_NORM	Negative normalized, nonzero *x*

Item	Description
FP_PLUS_DENORM	Positive denormalized, nonzero *x*
FP_MINUS_DENORM	Negative denormalized, nonzero *x*
FP_PLUS_ZERO	*x* = +0.0
FP_MINUS_ZERO	*x* = -0.0
FP_PLUS_INF	*x* = +INF
FP_MINUS_INF	*x* = -INF
FP_NANS	*x* = Signaling Not a Number (NaNS)
FP_NANQ	*x* = Quiet Not a Number (NaNQ)

Since class is a reserved keyword in C++, the **class** subroutine cannot be invoked in a C++ program. The **_class** subroutine is an interface for the C++ program using the **class** subroutine. The interface and the return values for **class** and **_class** subroutines are identical.

The **finite** subroutine returns a nonzero value if the *x* parameter is a finite number; that is, if *x* is not +-, INF, NaNQ, or NaNS.

The **isnan** subroutine returns a nonzero value if the *x* parameter is an NaNS or a NaNQ. Otherwise, it returns 0.

The **unordered** subroutine returns a nonzero value if a floating-point comparison between *x* and *y* would be unordered. Otherwise, it returns 0.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **-lm** flag. To compile the **class.c** file, for example, enter:

```
cc class.c -lm
```

Parameters

Item	Description
x	Specifies some double-precision floating-point value.
y	Specifies some double-precision floating-point value.

Error Codes

The **finite**, **isnan**, and **unordered** subroutines neither return errors nor set bits in the floating-point exception status, even if a parameter is an NaNS.

clear, erase, wclear or werase Subroutine

Purpose

Clears a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int clear(void);
```

```
int erase(void);
```

int wclear(WINDOW *win);

```
int werase(WINDOW *win);
```

Description

The **clear**, **erase**, **wclear**, and **werase** subroutines clear every position in the current or specified window.

The **clear** and **wclear** subroutines also achieve the same effect as calling the **clearok** subroutine, so that the window is cleared completely on the next call to the **wrefresh** subroutine for the window and is redrawn in its entirety.

Parameters

Item	Description
*win	Specifies the window to clear.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

For the **clear** and **wclear** subroutines:

1. To clear stdscr and set a clear flag for the next call to the **refresh** subroutine, enter:

   ```
   clear();
   ```

2. To clear the user-defined window `my_window` and set a clear flag for the next call to the **wrefresh** subroutine, enter:

   ```
   WINDOW *my_window;
   wclear(my_window);
   waddstr (my_window, "This will be cleared.");
   wrefresh (my_window);
   ```

3. To erase the standard screen structure, enter:

   ```
   erase();
   ```

4. To erase the user-defined window `my_window`, enter:

   ```
   WINDOW *my_window;
   werase (my_window);
   ```

   Note: After the **wrefresh**, the window will be cleared completely. You will not see the string "This will be cleared."

For the **erase** and **werase** subroutines:

1. To erase the standard screen structure, enter:

```
erase();
```

2. To erase the user-defined window `my_window`, enter:

```
WINDOW *my_window;
werase(my_window);
```

clearok, idlok, leaveok, scrollok, setscrreg or wsetscrreg Subroutine

Purpose

Terminal output control subroutines.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int clearok(WINDOW *win,
bool bf);

int idlok(WINDOW *win,
bool bf);

int leaveok(WINDOW *win,
bool bf);

int scrollok(WINDOW *win,
bool bf);

int setscrreg(int top,
int bot);

int wsetscrreg(WINDOW *win,
int top,
int bot);
```

Description

These subroutines set options that deal with output within Curses.

The **clearok** subroutine assigns the value of *bf* to an internal flag in the specified window that governs clearing of the screen during a refresh. If, during a refresh operation on the specified window, the flag in **curscr** is TRUE or the flag in the specified window is TRUE, then the implementation clears the screen, redraws it in its entirety, and sets the flag to FALSE in **curscr** and in the specified window. The initial state is unspecified.

The **idlok** subroutine specifies whether the implementation may use the hardware insert-line, delete-line, and scroll features of terminals so equIpped. If *bf* is TRUE, use of these features is enabled. If *bf* is FALSE, use of these features is disabled and lines are instead redrawn as required. The initial state is FALSE.

The **leaveok** subroutine controls the cursor position after a refresh operation. If *bf* is TRUE, refresh operations on the specified window may leave the terminal's cursor at an arbitrary position. If *bf* is FALSE, then at the end of any refresh operation, the terminal's cursor is positioned at the cursor position contained in the specified window. The initial state is FALSE.

The **scrollok** subroutine controls the use of scrolling. If *bf* is TRUE, then scrolling is enabled for the specified window, with the consequences discussed in Truncation, Wrapping and Scrolling on page 28. If *bf* is FALSE, scrolling is disabled for the specified window. The initial state is FALSE.

The **setscrreg** and **wsetscrreg** subroutines define a software scrolling region in the current or specified window. The *top* and *bot* arguments are the line numbers of the first and last line defining the scrolling region. (Line 0 is the top line of the window.) If this option and the **scrollok** subroutine are enabled, an attempt to move off the last line of the margin causes all lines in the scrolling region to scroll one line in the direction of the first line. Only characters in the window are scrolled. If a software scrolling region is set and the **scrollok** subroutine is not enabled, an attempt to move off the last line of the margin does not reposition any lines in the scrolling region.

Parameters

The parameters for the **clearok** subroutine are:

Item	Description
Flag	Sets the window clear flag. If TRUE, curses clears the window on the next call to the **wrefresh** or **refresh** subroutines. If FALSE, curses does not clear the window.
Window	Specifies the window to clear.

The parameters for the **idlok** subroutine are:

Item	Description
Flag	Specifies whether to enable curses to use the hardware insert/delete line feature (TRUE) or not (FALSE).
Window	Specifies the window it will affect.

The parameters for the **leaveok** subroutine are:

Item	Description
Flag	Specifies whether to leave the physical cursor alone after a refresh (TRUE) or to move the physical cursor to the logical cursor after a refresh (FALSE).
Window	Specifies the window for which to set the *Flag* parameter.

The parameters for the **scrollok** subroutine are:

Item	Description
Flag	Enables scrolling when set to TRUE. Otherwise, set the *Flag* parameter to FALSE to disable scrolling.
Window	Identifies the window in which to enable or disable scrolling.

The parameters for the **setscrreg** and **wsetscrreg** subroutines are:

Item	Description
Bmargin	Specifies the last line number in the scrolling region.
Tmargin	Specifies the first line number in the scrolling region (0 is the top line of the window.).
Window	Specifies the window in which to place the scrolling region. You specify this parameter only with the **wsetscrreg** subroutine.

Return Values

Upon successful completion, the **setscrreg** and **wsetscrreg** subroutines return OK. Otherwise, they return ERR.

The other subroutines always return OK.

Examples

Examples for the **clearok** subroutine are:

1. To set the user-defined screen `my_screen` to clear on the next call to the **wrefresh** subroutine, enter:

```
WINDOW *my_screen;
clearok(my_screen, TRUE);
```

2. To set the standard screen structure to clear on the next call to the **refresh** subroutine, enter:

```
clearok(stdscr, TRUE);
```

Examples for the **idlok** subroutine are:

1. To enable curses to use the hardware insert/delete line feature in stdscr, enter:

```
idlok(stdscr, TRUE);
```

2. To force curses not to use the hardware insert/delete line feature in the user-defined window `my_window` , enter:

```
idlok(my_window, FALSE);
```

Examples for the **leaveok** subroutine are:

1. To move the physical cursor to the same location as the logical cursor after refreshing the user-defined window `my_window`, enter:

```
WINDOW *my_window;
leaveok(my_window, FALSE);
```

2. To leave the physical cursor alone after refreshing the user-defined window `my_window`, enter:

```
WINDOW *my_window;
leaveok(my_window, TRUE);
```

Examples for the **scrollok** subroutine are:

1. To turn scrolling on in the user-defined window `my_window`, enter:

```
WINDOW *my_window;
scrollok(my_window, TRUE);
```

2. To turn scrolling off in the user-defined window `my_window`, enter:

```
WINDOW *my_window;
scrollok(my_window, FALSE);
```

Examples for the **setscrreg** or **wsetscrreg** subroutine are:

1. To set a scrolling region starting at the 10th line and ending at the 30th line in the stdscr, enter:

```
setscrreg(9, 29);
```

   Note: Zero is always the first line.

2. To set a scrolling region starting at the 10th line and ending at the 30th line in the user-defined window `my_window`, enter:

```
WINDOW *my_window;
wsetscrreg(my_window, 9, 29);
```

clrtobot or wclrtobot Subroutine

Purpose

Erases the current line from the logical cursor position to the end of the window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int clrtobot(void);
```

int wclrtobot(WINDOW **win*);

Description

The **clrtobot** and **wclrtobot** subroutines erase all lines following the cursor in the current or specified window, and erase the current line from the cursor to the end of the line, inclusive. These subroutines do not update the cursor.

Parameters

Item	Description
**win*	Specifies the window in which to erase lines.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To erase the lines below and to the right of the logical cursor in the stdscr, enter:

   ```
   clrtobot();
   ```

2. To erase the lines below and to the right of the logical cursor in the user-defined window `my_window`, enter:

   ```
   WINDOW *my_window;
   wclrtobot(my_window);
   ```

clrtoeol or wclrtoeol Subroutine

Purpose

Erases the current line from the logical cursor position to the end of the line.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int clrtoeol(void);
```

int wclrtoeol(WINDOW * *win*);

Description

The **clrtoeol** and **wclrtoeol** subroutines erase the current line from the cursor to the end of the line, inclusive, in the current or specified window. These subroutines do not update the cursor.

Parameters

Item	Description
**win*	Specifies the window in which to clear the line.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To clear the line to the right of the logical cursor in the stdscr, enter:

   ```
   clrtoeol();
   ```

2. To clear the line to the right of the logical cursor in the user-defined window `my_window`, enter:

   ```
   WINDOW *my_window;
   wclrtoeol(my_window);
   ```

clock Subroutine

Purpose

Reports central processing unit (CPU) time used.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>
```

```
clock_t clock (void);
```

Description

The **clock** subroutine reports the amount of CPU time used. The reported time is the sum of the CPU time of the calling process and its terminated child processes for which it has executed **wait**, **system**, or **pclose** subroutines. To measure the amount of time used by a program, the **clock** subroutine should be called at the beginning of the program, and that return value should be subtracted from the return value

of subsequent calls to the **clock** subroutine. To find the time in seconds, divide the value returned by the **clock** subroutine by the value of the macro **CLOCKS_PER_SEC**, which is defined in the **time.h** file.

Return Values

The **clock** subroutine returns the amount of CPU time used.

clock_getcpuclockid Subroutine

Purpose

Accesses a process CPU-time clock.

Syntax

```
#include <time.h>
int clock_getcpuclockid(pid_t pid, clockid_t *clock_id);
```

Description

The **clock_getcpuclockid** subroutine returns the clock ID of the CPU-time clock of the process specified by *pid*. If the process described by *pid* exists and the calling process has permission, the clock ID of this clock returns in *clock_id*.

If *pid* is zero, the **clock_getcpuclockid** subroutine returns the clock ID specified in *clock_id* of the CPU-time clock of the process making the call.

To obtain the CPU-time clock ID of other processes, the calling process should be root or have the same effective or real user ID as the process that owns the targetted CPU-time clock.

Parameters

Item	Description
clock_id	Specifies the clock ID of the CPU-time clock.
pid	Specifies the process ID of the CPU-time clock.

Return Values

Upon successful completion, the **clock_getcpuclockid** subroutine returns 0; otherwise, an error code is returned indicating the error.

Error Codes

Item	Description
ENOTSUP	The function is not supported with checkpoint-restart processes.
EPERM	The requesting process does not have permission to access the CPU-time clock for the process.
ESRCH	No process can be found corresponding to the process specified by *pid*.

clock_getres, clock_gettime, and clock_settime Subroutine

Purpose

Clock and timer functions.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>

int clock_getres (clock_id, res)
clockid_t clock_id;
struct timespec *res;

int clock_gettime (clock_id, tp)
clockid_t clock_id;
struct timespec *tp;

int clock_settime (clock_id, tp)
clockid_t clock_id;
const struct timespec *tp;
```

Description

The **clock_getres** subroutine returns the resolution of any clock. Clock resolutions are implementation-defined and cannot be set by a process. If the *res* parameter is not NULL, the resolution of the specified clock is stored in the location pointed to by the *res* parameter. If the *res* parameter is NULL, the clock resolution is not returned. If the *time* parameter of the **clock_settime** subroutine is not a multiple of the *res* parameter, the value is truncated to a multiple of the *res* parameter.

The **clock_gettime** subroutine returns the current value, *tp*, for the specified clock, *clock_id*.

The **clock_settime** subroutine sets the specified clock, *clock_id*, to the value specified by the *tp* parameter. Time values that are between two consecutive non-negative integer multiples of the resolution of the specified clock will be truncated down to the smaller multiple of the resolution.

A clock may be system-wide (visible to all processes) or per-process (measuring time that is meaningful only within a process). All implementations support a *clock_id* of **CLOCK_REALTIME** as defined in the **time.h** file. This clock represents the Realtime clock for the system. For this clock the values returned by the **clock_gettime** subroutine and specified by the **clock_settime** subroutine represent the amount of time (in seconds and nanoseconds) since the epoch.

If the value of the **CLOCK_REALTIME** clock is set through the **clock_settime** subroutine, the new value of the clock is used to determine the time of expiration for absolute time services based upon the **CLOCK_REALTIME** clock. This applies to the time at which armed absolute timers expire. If the absolute time requested at the invocation of such a time service is before the new value of the clock, the time service expires immediately as if the clock had reached the requested time normally.

Setting the value of the **CLOCK_REALTIME** clock through the **clock_settime** subroutine has no effect on threads that are blocked waiting for a relative time service based upon this clock, including the **nanosleep** subroutine; nor on the expiration of relative timers based upon this clock. Consequently, these time services expire when the requested relative interval elapses, independently of the new or old value of the clock.

A *clock_id* of **CLOCK_MONOTONIC** is defined in the **time.h** file. This clock represents the monotonic clock for the system. For this clock, the value returned by the **clock_gettime** subroutine represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past. This point does not change after system start time (for example, this clock cannot have backward jumps). The value of the

CLOCK_MONOTONIC clock cannot be set through the **clock_settime** subroutine. This subroutine fails if it is invoked with a *clock_id* parameter of **CLOCK_MONOTONIC**.

The calling process should have **SYS_OPER** authority to set the value of the **CLOCK_REALTIME** clock.

Process CPU-time clocks are supported by the system. For these clocks, the values returned by **clock_gettime** and specified by **clock_settime** represent the amount of execution time of the process associated with the clock. **Clockid_t** values for CPU-time clocks are obtained by calling **clock_getcpuclockid**. A special **clockid_t** value, **CLOCK_PROCESS_CPUTIME_ID**, is defined in the **time.h** file. This value represents the CPU-time clock of the calling process when one of the **clock_*** or **timer_*** functions is called.

To get or set the value of a CPU-time clock, the calling process must have root permissions or have the same effective or real user ID as the process that owns the targeted CPU-time clock. The same rule applies to a process that tries to get the resolution of a CPU-time clock.

Thread CPU-time clocks are supported by the system. For these clocks, the values returned by **clock_gettime** and specified by **clock_settime** represent the amount of execution time of the thread associated with the clock. **Clockid_t** values for thread CPU-time clocks are obtained by calling the **pthread_getcpuclockid** subroutine. A special **clockid_t** value, **CLOCK_THREAD_CPUTIME_ID**, is defined in the **time.h** file. This value represents the thread CPU-time clock of the calling thread when one of the **clock_*()** or **timer_*** functions is called.

To get or set the value of a thread CPU-time clock, the calling thread must be a thread in the same process as the one that owns the targeted thread CPU-time clock. The same rule applies to a thread that tries to get the resolution of a thread CPU-time clock.

Parameters

Item	Description
clock_id	Specifies the clock.
res	Stores the resolution of the specified clock.
tp	Stores the current value of the specified clock.

Return Values

If successful, 0 is returned. If unsuccessful, -1 is returned, and **errno** will be set to indicate the error.

Error Codes

The **clock_getres**, **clock_gettime**, and **clock_settime** subroutines fail if:

Item	Description
EINVAL	The *clock_id* parameter does not specify a known clock.
ENOTSUP	The function is not supported with checkpoint-restart processes.

The **clock_settime** subroutine fails if:

Item	Description
EINVAL	The *tp* parameter to the **clock_settime** subroutine is outside the range for the given clock ID.
EINVAL	The *tp* parameter specified a nanosecond value less than zero or greater than or equal to 1000 million.
EINVAL	The value of the *clock_id* argument is **CLOCK_MONOTONIC**.

The **clock_settime** subroutine might fail if:

Item	Description
EPERM	The requesting process does not have the appropriate privilege to set the specified clock.

clock_nanosleep Subroutine

Purpose

Specifies clock for high resolution sleep.

Syntax

```
#include <time.h>
 int clock_nanosleep(clockid_t clock_id, int flags,
       const struct timespec *rqtp, struct timespec *rmtp);
```

Description

If the **TIMER_ABSTIME** flag is not set in the *flags* argument, the **clock_nanosleep** subroutine causes the current thread to be suspended from execution until either the time interval specified by the *rqtp* argument has elapsed, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. The *clock_id* argument specifies the clock used to measure the time interval.

If the **TIMER_ABSTIME** flag is set in the *flags* argument, the **clock_nanosleep** subroutine causes the current thread to be suspended from execution until either the time value of the clock specified by *clock_id* reaches the absolute time specified by the *rqtp* argument, or a signal is delivered to the calling thread and its action is to invoke a signal-catching function, or the process is terminated. If, at the time of the call, the time value specified by *rqtp* is less than or equal to the time value of the specified clock, then the **clock_nanosleep** subroutine returns immediately and the calling process shall not be suspended.

The suspension time caused by this function might be longer than requested either because the argument value is rounded up to an integer multiple of the sleep resolution, or because of the scheduling of other activity by the system. Except for the case of being interrupted by a signal, the suspension time for the relative **clock_nanosleep** subroutine (that is, with the TIMER_ABSTIME flag not set) shall not be less than the time interval specified by the *rqtp* argument, as measured by the corresponding clock. The suspension for the absolute **clock_nanosleep** subroutine (that is, with the TIMER_ABSTIME flag set) is in effect at least until the value of the corresponding clock reaches the absolute time specified by the *rqtp* argument, except for the case of being interrupted by a signal.

The **clock_nanosleep** subroutine has no effect on the action or blocking of any signal.

The subroutine fails if the *clock_id* argument refers to a process or a thread CPU-time clock.

Parameters

Item	Description
clock_id	Specifies the clock used to measure the time.
flags	Identifies the type of timeout. If **TIMER_ABSTIME** is set, the time value pointed to by *rqtp* is an absolute time value; otherwise, it is a time interval.
rmtp	Points to the **timespec** structure used to return the remaining amount of time in an interval (the requested time minus the time actually slept) if the sleep is interrupted.
rqtp	Points to the **timespec** structure that contains requested sleep time.

Return Values

The **clock_nanosleep** subroutine returns 0 when the requested time has elapsed.

The **clock_nanosleep** subroutine returns the corresponding error value when it has been interrupted by a signal. For the relative **clock_nanosleep**subroutine, when the *rmtp* argument is not NULL, the referenced **timespec** structure is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* argument is NULL, the remaining time is not returned. The absolute **clock_nanosleep** subroutine has no effect on the structure referenced by the *rmtp* argument.

Error Codes

Item	Description
EINTR	The **clock_nanosleep** subroutine was interrupted by a signal.
EINVAL	The *rqtp* parameter specified a nanosecond value less than 0 or greater than or equal to 1000 million; or the **TIMER_ABSTIME** flag was specified in the **flags** parameter and the *rqtp* parameter is outside the range for the clock specified by *clock_id*; or the *clock_id* parameter does not specify a known clock, or specifies the CPU-time clock of the calling thread.
ENOTSUP	The *clock_id* argument specifies a clock for which the *clock_nanosleep* subroutine is not supported, such as a CPU-time clock.
ENOTSUP	The subroutine is not supported with checkpoint-restarted processes.

Files

timer.h

clog, clogf, or clogl Subroutine

Purpose

Computes the complex natural logarithm.

Syntax

```
#include <complex.h>

double complex clog (z)
double complex z;

float complex clogf (z)
float complex z;

long double complex clogl (z)
long double complex z;
```

Description

The **clog**, **clogf**, and **clogl** subroutines compute the complex natural (base *e*) logarithm of *z*, with a branch cut along the negative real axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **clog**, **clogf**, and **clogl** subroutines return the complex natural logarithm value, in the range of a strip mathematically unbounded along the real axis and in the interval [-*i* pi, +*i* pi] along the imaginary axis.

close Subroutine

Purpose

Closes a file descriptor.

Syntax

```
#include <unistd.h>

int close (
 FileDescriptor)
int FileDescriptor;
```

Description

The **close** subroutine closes the file or shared memory object associated with the *FileDescriptor* parameter. If Network File System (NFS) is installed on your system, this file can reside on another node.

All file regions associated with the file specified by the *FileDescriptor* parameter that this process has previously locked with the **lockf** or **fcntl** subroutine are unlocked. This occurs even if the process still has the file open by another file descriptor.

If the *FileDescriptor* parameter resulted from an **open** subroutine that specified **O_DEFER**, and this was the last file descriptor, all changes made to the file since the last **fsync** subroutine are discarded.

If the *FileDescriptor* parameter is associated with a mapped file, it is unmapped. The **shmat** subroutine provides more information about mapped files.

The **close** subroutine attempts to cancel outstanding **asynchronous I/O requests** on this file descriptor. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.

If the *FileDescriptor* parameter is associated with a shared memory object and the shared memory object remains referenced at the last close (that is, a process has it mapped), the entire contents of the memory object persists until the memory object becomes unreferenced. If this is the last close of a shared memory object and the close results in the memory object becoming unreferenced, and the memory object has been unlinked, the memory object is removed. The **shm_open** subroutine provides more information about shared memory objects.

The **close** subroutine is blocked until all subroutines which use the file descriptor return to **usr** space. For example, when a thread is calling **close** and another thread is calling **select** with the same file descriptor, the **close** subroutine does not return until the **select** call returns.

When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO is discarded. If the link count of the file is 0 when all file descriptors associated with the file have been closed, the space occupied by the file is freed, and the file is no longer accessible.

Note: If the *FileDescriptor* parameter refers to a device and the **close** subroutine actually results in a device **close**, and the device **close** routine returns an error, the error is returned to the application. However, the *FileDescriptor* parameter is considered closed and it may not be used in any subsequent calls.

All open file descriptors are closed when a process exits. In addition, file descriptors may be closed during the **exec** subroutine if the **close-on-exec** flag has been set for that file descriptor.

Parameters

Item	Description
FileDescriptor	Specifies a valid open file descriptor.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to identify the error.

The underlying file system implementation might report any one of the values from the `/usr/include/errno.h` file to the **close** subroutine. The **close** subroutine returns a value of -1 and the **errno** global variable is set to the return value from the file system, but the file is still closed. The state of the *FileDescriptor* parameter is closed except for the conditions specified in the **Error Codes** section.

Error Codes

The **close** subroutine is unsuccessful if the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid open file descriptor.

The **close** subroutine may also be unsuccessful if the file being closed is NFS-mounted and the server is down under the following conditions:

- The file is on a hard mount.
- The file is locked in any manner.

The **close** subroutine may also be unsuccessful if NFS is installed and the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

The success of the **close** subroutine is undetermined if the following is true:

Item	Description
EINTR	The state of the *FileDescriptor* is undetermined. Retry the **close** routine to ensure that the *FileDescriptor* is closed.

cnd_broadcast, cnd_destroy, cnd_init, cnd_signal, cnd_timedwait and cnd_wait Subroutine

Purpose

The **cnd_broadcast** subroutine unblocks all the threads that are blocked by using the *cond* condition variable.

The **cnd_destroy** subroutine releases all the resources that are used by the *cond* condition variable.

The **cnd_init** subroutine creates a *cond* condition variable.

The **cnd_signal** subroutine unblocks one of the threads that is blocked by using the condition that is specified by the *cond* parameter.

The **cnd_timedwait** subroutine unblocks the condition that is specified by the *cond* condition variable after a specified time indicated by the **ts** parameter.

The **cnd_wait** subroutine blocks the condition that is specified by the *cond* condition variable until it gets a signal from the **cnd_signal** or **cnd_broadcast** subroutines.

Library

Standard C library (**libc.a**)

Syntax

```
#include <threads.h>
int cnd_broadcast (cnd_t * cond);

void cnd_destroy (cnd_t * cond);

int cnd_init (cnd_t * cond);

int cnd_signal (cnd_t * cond);

int cnd_timedwait (cnd_t * restrict cond, mtx_t * restrict mtx, const struct timespec *
restrict ts);

int cnd_wait (cnd_t * cond, mtx_t * mtx);
```

Description

The **cnd_broadcast** subroutine unblocks all the threads that are blocked by using the condition variable specified by the **cond** parameter during the function call.

If no threads are blocked by using the condition variable specified by the **cond** parameter during the function call, the function is inactive.

The **cnd_destroy** subroutine releases all the resources that are used by the condition variable specified by the **cond** parameter.

The **cnd_destroy** subroutine requires that threads are not blocked while waiting for the condition variable specified by the **cond** parameter.

The **cnd_init** subroutine creates a condition variable. If the subroutine is successful, it sets the variable specified by the **cond** parameter to a value that uniquely identifies the newly created condition variable.

A thread that calls the **cnd_wait** subroutine on a newly created condition variable is blocked.

The **cnd_signal** subroutine unblocks one of the threads that are blocked by using the condition variable specified by the **cond** parameter during the function call. If threads are not blocked by using the condition variable during the function call, the function is inactive and returns success.

The **cnd_timedwait** and **cnd_wait** subroutine automatically unlocks and locks the mutex specified by the **mtx** parameter and tries to block until the condition variable pointed to by the **cond** is signaled by a call to the **cnd_signal** or **cnd_broadcast** subroutine, or until the **TIME_UTC** based calendar time is specified by the value of the **ts** parameter.

When the calling thread is unblocked, it locks the variable specified by the **mtx** parameter before it returns a value. The **cnd_timedwait** subroutine requires that the mutex specified by the **mtx** parameter is locked by the calling thread.

Parameters

Item	Description
cond	Specifies the condition variable to be created or released, depending on the type of the subroutine in which the parameter is referenced.
mtx	Specifies the mutex to be unlocked.
ts	Specifies the maximum time for the condition variable to be blocked.

Return Values

The **cnd_broadcast**, **cnd_signal**, and **cnd_wait** subroutine returns the value of **thrd_success** on success, and returns the value of **thrd_error** if the request cannot be processed.

The **cnd_destroy** subroutine returns no value.

The **cnd_init** subroutine returns the value of **thrd_success** on success.

The **cnd_init** subroutine returns the value of **thrd_nomem** if memory cannot be allocated for the newly created condition, and returns the value of **thrd_error** if the request cannot be processed.

The **cnd_timedwait** subroutine returns the value of **thrd_success** on success, or returns the value of **thrd_timedout** if the time specified in the call is reached without acquiring the requested resource, and returns the value of **thrd_error** if the request cannot be processed.

Files

The **threads.h** file defines standard macros, data types, and subroutines.

compare_and_swap and compare_and_swaplp Subroutines

Purpose

Conditionally updates or returns a variable atomically.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/atomic_op.h>
boolean_t compare_and_swap ( addr, old_val_addr, new_val)
atomic_p addr;
int *old_val_addr;
int new_val;

boolean_t compare_and_swaplp ( addr, old_val_addr, new_val)
atomic_l addr;
long *old_val_addr;
long new_val;
```

Description

The **compare_and_swap** and **compare_and_swaplp** subroutines perform an atomic operation that compares the contents of a variable with a stored old value. If the values are equal, a new value is stored in the variable and **TRUE** is returned. If the values are not equal, the old value is set to the current value of the variable and **FALSE** is returned.

For 32-bit applications, the **compare_and_swap** and **compare_and_swaplp** subroutines are identical and operate on a word aligned single word (32-bit variable aligned on a 4-byte boundary).

For 64-bit applications, the **compare_and_swap** subroutine operates on a word aligned single word (32-bit variable aligned on a 4-byte boundary) and the **compare_and_swaplp** subroutine operates on a double word aligned double word (64-bit variable aligned on an 8-byte boundary).

The **compare_and_swap** and **compare_and_swaplp** subroutines are useful when a word value must be updated only if it has not been changed since it was last read.

Note: If the **compare_and_swap** or the **compare_and_swaplp** subroutine is used as a locking primitive, insert an **isync** at the start of any critical sections.

Parameters

Item	Description
addr	Specifies the address of the variable.
old_val_addr	Specifies the address of the old value to be checked against (and conditionally updated with) the value of the variable.
new_val	Specifies the new value to be conditionally assigned to the variable.

Return Values

Item	Description
TRUE	Indicates that the variable was equal to the old value, and has been set to the new value.
FALSE	Indicates that the variable was not equal to the old value, and that its current value has been returned to the location where the old value was previously stored.

compile, step, or advance Subroutine

Purpose

Compiles and matches regular-expression patterns.

Note: Commands use the **regcomp**, **regexec**, **regfree**, and **regerror** subroutines for the functions described in this article.

Library

Standard C Library (**libc.a**)

Syntax

```
#define INIT declarations
#define GETC( ) getc_code
#define PEEKC( ) peekc_code
#define UNGETC(c) ungetc_code
#define RETURN(pointer) return_code
#define ERROR(val) error_code
```

```
#include <regexp.h>
#include <NLregexp.h>
```

```
char *compile (InString, ExpBuffer, EndBuffer, EndOfFile)
char * ExpBuffer;
char  * InString, * EndBuffer;
int  EndOfFile;

int step (String, ExpBuffer)
const char * String, *ExpBuffer;
```

```
int advance (String, ExpBuffer)
const char *String, *ExpBuffer;
```

Description

The **/usr/include/regexp.h** file contains subroutines that perform regular-expression pattern matching. Programs that perform regular-expression pattern matching use this source file. Thus, only the **regexp.h** file needs to be changed to maintain regular expression compatibility between programs.

The interface to this file is complex. Programs that include this file define the following six macros before the **#include <regexp.h>** statement. These macros are used by the **compile** subroutine:

Item	Description
INIT	This macro is used for dependent declarations and initializations. It is placed right after the declaration and opening { (left brace) of the **compile** subroutine. The definition of the **INIT** buffer must end with a ; (semicolon). **INIT** is frequently used to set a register variable to point to the beginning of the regular expression so that this register variable can be used in the declarations for the **GETC**, **PEEKC**, and **UNGETC** macros. Otherwise, you can use **INIT** to declare external variables that **GETC**, **PEEKC**, and **UNGETC** require.
GETC( )	This macro returns the value of the next character in the regular expression pattern. Successive calls to the **GETC** macro should return successive characters of the pattern.
PEEKC( )	This macro returns the next character in the regular expression. Successive calls to the **PEEKC** macro should return the same character, which should also be the next character returned by the **GETC** macro.
UNGETC(*c***)**	This macro causes the parameter *c* to be returned by the next call to the **GETC** and **PEEKC** macros. No more than one character of pushback is ever needed, and this character is guaranteed to be the last character read by the **GETC** macro. The return value of the **UNGETC** macro is always ignored.
RETURN(*pointer***)**	This macro is used for normal exit of the **compile** subroutine. The *pointer* parameter points to the first character immediately following the compiled regular expression. This is useful for programs that have memory allocation to manage.

Item	Description
ERROR(*val***)**	This macro is used for abnormal exit from the **compile** subroutine. It should never contain a **return** statement. The *val* parameter is an error number. The error values and their meanings are: **Error** **Meaning** **11** Interval end point too large **16** Bad number **25** \ *digit* out of range **36** Illegal or missing delimiter **41** No remembered search String **42** \ (?\) imbalance **43** Too many \.(**44** More than two numbers given in \{ \} **45** } expected after \. **46** First number exceeds second in \{ \} **49** [ ] imbalance **50** Regular expression overflow **70** Invalid endpoint in range

The **compile** subroutine compiles the regular expression for later use. The *InString* parameter is never used explicitly by the **compile** subroutine, but you can use it in your macros. For example, you can use the **compile** subroutine to pass the string containing the pattern as the *InString* parameter to **compile** and use the **INIT** macro to set a pointer to the beginning of this string. The example in the "Examples" on page 189 section uses this technique. If your macros do not use *InString*, then call **compile** with a value of **((char *) 0)** for this parameter.

The *ExpBuffer* parameter points to a character array where the compiled regular expression is to be placed. The *EndBuffer* parameter points to the location that immediately follows the character array where the compiled regular expression is to be placed. If the compiled expression cannot fit in (*EndBuffer-ExpBuffer*) bytes, the call **ERROR(***50***)** is made.

The *EndOfFile* parameter is the character that marks the end of the regular expression. For example, in the **ed** command, this character is usually / (slash).

The **regexp.h** file defines other subroutines that perform actual regular-expression pattern matching. One of these is the **step** subroutine.

The *String* parameter of the **step** subroutine is a pointer to a null-terminated string of characters to be checked for a match.

The *Expbuffer* parameter points to the compiled regular expression, obtained by a call to the **compile** subroutine.

The **step** subroutine returns the value 1 if the given string matches the pattern, and 0 if it does not match. If it matches, then **step** also sets two global character pointers: **loc1**, which points to the first character that matches the pattern, and **loc2**, which points to the character immediately following the last character that matches the pattern. Thus, if the regular expression matches the entire string, **loc1** points to the first character of the *String* parameter and **loc2** points to the null character at the end of the *String* parameter.

The **step** subroutine uses the global variable **circf**, which is set by the **compile** subroutine if the regular expression begins with a ^ (circumflex). If this variable is set, **step** only tries to match the regular expression to the beginning of the string. If you compile more than one regular expression before executing the first one, save the value of **circf** for each compiled expression and set **circf** to that saved value before each call to **step**.

Using the same parameters that were passed to it, the **step** subroutine calls a subroutine named **advance**. The **step** function increments through the *String* parameter and calls the **advance** subroutine until it returns a 1, indicating a match, or until the end of *String* is reached. To constrain the *String* parameter to the beginning of the string in all cases, call the **advance** subroutine directly instead of calling the **step** subroutine.

When the **advance** subroutine encounters an * (asterisk) or a \{ \} sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself, trying to match the rest of the string to the rest of the regular expression. As long as there is no match, the **advance** subroutine backs up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. You can stop this backing-up before the initial point in the string is reached. If the **locs** global character is equal to the point in the string sometime during the backing-up process, the **advance** subroutine breaks out of the loop that backs up and returns 0. This is used for global substitutions on the whole line so that expressions such as `s/y*//g` do not loop forever.

Note: In 64-bit mode, these interfaces are not supported: they fail with a return code of 0. In order to use the 64-bit version of this functionality, applications should migrate to the **fnmatch**, **glob**, **regcomp**, and **regexec** functions which provide full internationalized regular expression functionality compatible with ISO 9945-1:1996 (IEEE POSIX 1003.1) and with the UNIX98 specification.

Parameters

Item	Description
InString	Specifies the string containing the pattern to be compiled. The *InString* parameter is not used explicitly by the **compile** subroutine, but it may be used in macros.
ExpBuffer	Points to a character array where the compiled regular expression is to be placed.
EndBuffer	Points to the location that immediately follows the character array where the compiled regular expression is to be placed.
EndOfFile	Specifies the character that marks the end of the regular expression.
String	Points to a null-terminated string of characters to be checked for a match.

Examples

The following is an example of the regular expression macros and calls:

```
#define INIT         register char *sp=instring;
#define GETC()         (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)      (--sp)
#define RETURN(c)      return;
#define ERROR(c)      regerr()

#include <regexp.h>
 . . .
compile (patstr,expbuf, &expbuf[ESIZE], '\0');
```

```
 . . .
if (step (linebuf, expbuf))
   succeed( );
 . . .
```

confstr Subroutine

Purpose

Gets configurable variables.

Library

Standard C library (**libc.a**)

Syntax

#include <unistd.h>

size_t confstr (int *name*, *char* ***** *buf*, size_t *len* **);**

Description

The **confstr** subroutine determines the current setting of certain system parameters, limits, or options that are defined by a string value. It is mainly used by applications to find the system default value for the **PATH** environment variable. Its use and purpose are similar to those of the **sysconf** subroutine, but it returns string values rather than numeric values.

If the *Len* parameter is not 0 and the *Name* parameter has a system-defined value, the **confstr** subroutine copies that value into a *Len*-byte buffer pointed to by the *Buf* parameter. If the string returns a value longer than the value specified by the *Len* parameter, including the terminating null byte, then the **confstr** subroutine truncates the string to *Len*-1 bytes and adds a terminating null byte to the result. The application can detect that the string was truncated by comparing the value returned by the **confstr** subroutine with the value specified by the *Len* parameter.

Parameters

Item	Description
Name	Specifies the system variable setting to be returned. Valid values for the *Name* parameter are defined in the **unistd.h** file.
Buf	Points to the buffer into which the **confstr** subroutine copies the value of the *Name* parameter.
Len	Specifies the size of the buffer storing the value of the *Name* parameter.

Return Values

If the value specified by the *Name* parameter is system-defined, the **confstr** subroutine returns the size of the buffer needed to hold the entire value. If this return value is greater than the value specified by the *Len* parameter, the string returned as the *Buf* parameter is truncated.

If the value of the *Len* parameter is set to 0 and the *Buf* parameter is a null value, the **confstr** subroutine returns the size of the buffer needed to hold the entire system-defined value, but does not copy the string value. If the value of the *Len* parameter is set to 0 but the *Buf* parameter is not a null value, the result is unspecified.

Error Codes

The **confstr** subroutine will fail if:

Item	Description
EINVAL	The value of the name argument is invalid.

Example

To find out what size buffer is needed to store the string value of the *Name* parameter, enter:

```
confstr(_CS_PATH, NULL, (size_t) 0)
```

The **confstr** subroutine returns the size of the buffer.

Files

Item	Description
/usr/include/limits.h	Contains system-defined limits.
/usr/include/unistd.h	Contains system-defined environment variables.

conj, conjf, or conjl Subroutine

Purpose

Computes the complex conjugate.

Syntax

```
#include <complex.h>

double complex conj (z)
double complex z;

float complex conjf (z)
float complex z;

long double complex conjl (z)
long double complex z;
```

Description

The **conj**, **conjf**, or **conjl** subroutines compute the complex conjugate of *z*, by reversing the sign of its imaginary part.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **conj**, **conjf**, or **conjl** subroutines return the complex conjugate value.

color_content Subroutine

Purpose

Returns the current intensity of the red, green, and blue (RGB) components of a color.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
color_content(Color,  R, G,
B)
short   Color;
short  *R, * G, * B;
```

Description

The **color_content** subroutine, given a color number, returns the current intensity of its red, green, and blue (RGB) components. This subroutine stores the information in the address specified by the *R*, *G*, and *B* arguments. If successful, this returns OK. Otherwise, this subroutine returns ERR if the color does not exist, is outside the valid range, or the terminal cannot change its color definitions.

To determine if you can change the color definitions for a terminal, use the **can_change_color** subroutine. You must call the **start_color** subroutine before you can call the **color_content** subroutine.

Note: The values stored at the addresses pointed to by *R*, *G*, and *B* are between 0 (no component) and 1000 (maximum amount of component) inclusive.

Return Values

Item	Description
OK	Indicates the subroutine was successful.
ERR	Indicates the color does not exist, is outside the valid range, or the terminal cannot change its color definitions.

Parameters

Item	Description
B	Points to the address that stores the intensity value of the blue component.
Color	Specifies the color number. The color parameter must be a value between **0** and **COLORS-1** inclusive.
R	Points to the address that stores the intensity value of the red component.
G	Points to the address that stores the intensity value of the green component.

Example

To obtain the RGB component information for color 10 (assuming the terminal supports at least 11 colors), use:

```
short *r, *g, *b; color_content(10,r,g,b);
```

conv Subroutines

Purpose

Translates characters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ctype.h>

int toupper ( Character)
int Character;

int tolower (Character)
int Character;

int _toupper (Character)
int Character;

int _tolower (Character)
int Character;

int toascii (Character)
int Character;

int NCesc ( Pointer,  CharacterPointer)
NLchar *Pointer;
char *CharacterPointer;

int NCtoupper ( Xcharacter)
int Xcharacter;

int NCtolower (Xcharacter)
int Xcharacter;

int _NCtoupper (Xcharacter)
int Xcharacter;

int _NCtolower (Xcharacter)
int Xcharacter;

int NCtoNLchar (Xcharacter)
int Xcharacter;

int NCunesc (CharacterPointer, Pointer)
char *CharacterPointer;
NLchar *Pointer;

int NCflatchr (Xcharacter)
int Xcharacter;
```

Description

The **toupper** and the **tolower** subroutines have as domain an **int**, which is representable as an unsigned **char** or the value of **EOF**: -1 through 255.

If the parameter of the **toupper** subroutine represents a lowercase letter and there is a corresponding uppercase letter (as defined by **LC_CTYPE**), the result is the corresponding uppercase letter. If the parameter of the **tolower** subroutine represents an uppercase letter, and there is a corresponding lowercase letter (as defined by **LC_CTYPE**), the result is the corresponding lowercase letter. All other values in the domain are returned unchanged. If case-conversion information is not defined in the current locale, these subroutines determine character case according to the "C" locale.

The **_toupper** and **_tolower** subroutines accomplish the same thing as the **toupper** and **tolower** subroutines, but they have restricted domains. The **_toupper** routine requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. The **_tolower** routine requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The **NC***xxxxxx* subroutines translate all characters, including extended characters, as code points. The other subroutines translate traditional ASCII characters only. The **NC***xxxxxx* subroutines are obsolete and should not be used if portability and future compatibility are a concern.

The value of the *Xcharacter* parameter is in the domain of any legal **NLchar** data type. It can also have a special value of -1, which represents the end of file **(EOF)**.

If the parameter of the **NCtoupper** subroutine represents a lowercase letter according to the current collating sequence configuration, the result is the corresponding uppercase letter. If the parameter of the **NCtolower** subroutine represents an uppercase letter according to the current collating sequence configuration, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The **_NCtoupper** and **_NCtolower** routines are macros that perform the same function as the **NCtoupper** and **NCtolower** subroutines, but have restricted domains and are faster. The **_NCtoupper** macro requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. The **_NCtolower** macro requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The **NCtoNLchar** subroutine yields the value of its parameter with all bits turned off that are not part of an **NLchar** data type.

The **NCesc** subroutine converts the **NLchar** value of the *Pointer* parameter into one or more ASCII bytes stored in the character array pointed to by the *CharacterPointer* parameter. If the **NLchar** data type represents an extended character, it is converted into a printable ASCII escape sequence that uniquely identifies the extended character. **NCesc** returns the number of bytes it wrote. The display symbol table lists the escape sequence for each character.

The opposite conversion is performed by the **NCunesc** macro, which translates an ordinary ASCII byte or escape sequence starting at *CharacterPointer* into a single **NLchar** at *Pointer*. **NCunesc** returns the number of bytes it read.

The **NCflatchr** subroutine converts its parameter value into the single ASCII byte that most closely resembles the parameter character in appearance. If no ASCII equivalent exists, it converts the parameter value to a ? (question mark).

Note: The **setlocale** subroutine may affect the conversion of the decimal point symbol and the thousands separator.

Parameters

Item	Description
Character	Specifies the character to be converted.
Xcharacter	Specifies an **NLchar** value to be converted.

Item	Description
CharacterPointer	Specifies a pointer to a single-byte character array.
Pointer	Specifies a pointer to an escape sequence.

copysign, copysignf, copysignl , copysignd32, copysignd64, and copysignd128 Subroutines

Purpose

Perform number manipulation.

Syntax

```
#include <math.h>

double copysign (x, y)
double x, double y;

float copysignf (x, y)
float x, float y;

long double copysignl (x, y)
long double x, long double y;

_Decimal32 copysignd32(x, y)
_Decimal32 x;
_Decimal32 y;

_Decimal64 copysignd64(x, y)
_Decimal64 x;
_Decimal64 y;

_Decimal128 copysignd128(x, y)
_Decimal128 x;
_Decimal128 y;
```

Description

The **copysign**, **copysignf**, **copysignl**, **copysignd32**, **copysignd64**, and **copysignd128** subroutines produce a value with the magnitude of *x* and the sign of *y*.

Parameters

Item	Description
x	Specifies the magnitude.
y	Specifies the sign.

Return Values

Upon successful completion, the **copysign**, **copysignf**, **copysignl**, **copysignd32**, **copysignd64**, and **copysignd128** subroutines return a value with a magnitude of *x* and a sign of *y*.

copywin Subroutine

Purpose

Copies a region of a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int copywin(const WINDOW  *scrwin,
WINDOW  *dstwin,
int  sminrow,
int  smincol,
int  dminrow,
int  dmincol,
int  dmaxrow,
int  dmaxcol,
int  overlay);
```

Description

The **copywin** subroutine provides a finer granularity of control over the **overlay** and **overwrite** subroutines. As in the **prefresh** subroutine, a rectangle is specified in the destination window, (*dimrow*, *dimincol*) and (*dmaxrow*, *dmaxcol*), and the upper-left-corner coordinates of the source window, (*sminrow*, *smincol*). If the **overlay** subroutine is TRUE, then copying is non-destructive, as in the **overlay** subroutine. If the **overlay** subroutine is FALSE, then copying is destructive, as in the **overwrite** subroutine.

Parameters

Item	Description
**srcwin*	Points to the source window containing the region to copy.
**dstwin*	Points to the destination window to copy into.
sminrow	Specifies the upper left row coordinate of the source region.
smincol	Specifies the upper left column coordinate of the source region.
dminrow	Specifies the upper left row coordinate of the destination region.
dmincol	Specifies the upper left column coordinate for the destination region.
dmaxrow	Specifies the lower right row coordinate for the destination region.
dmaxcol	Specifies the lower right column coordinate for the destination region.
overlay	Sets the type of copy. If set to TRUE the copy is nondestructive. Otherwise, if set to FALSE, the copy is destructive.

Return Values

Upon successful completion, the **copywin** subroutine returns OK. Otherwise, it returns ERR.

Examples

To copy to an area in the destination window defined by coordinates (30,40), (30,49), (39,40), and (39,49) beginning with coordinates (0,0) in the source window, enter the following:

```
WINDOW *srcwin, *dstwin;

copywin(srcwin, dstwin,
```

```
 0, 0, 30,40, 39, 49,
  TRUE);
```

The example copies ten rows and ten columns from the source window beginning with coordinates (0,0) to the region in the destination window defined by the upper left coordinates (30, 40) and lower right coordinates (39, 49). Because the Overlay parameter is set to TRUE, the copy is nondestructive and blanks from the source window are not copied.

coredump Subroutine

Purpose

Creates a **core** file without terminating the calling process.

Library

Standard C library (**libc.a**)

Syntax

```
#include <core.h>

int coredump( coredumpinfop)
struct coredumpinfo *coredumpinfop ;
```

Description

The **coredump** subroutine creates a **core** file of the calling process without terminating the calling process. The created **core** file contains the memory image of the process, and this can be used with the **dbx** command for debugging purposes. In multithreaded processes, only one thread at a time should attempt to call this subroutine. Subsequent calls to **coredump** while a core dump (initiated by another thread) is in progress will fail.

Applications expected to use this facility need to be built with the **-bM:UR binder** flag, otherwise the routine will fail with an error code of **ENOTSUP**.

The **coredumpinfo** structure has the following fields:

Member Type	Member Name	Description
unsigned int	length	Length of the **core** file name
char *	name	Points to a character string that contains the name of the **core** file
int	reserved[8]	Reserved fields for future use

Parameters

Item	Description
coredumpinfop	Points to the **coredumpinfo** structure

If a NULL pointer is passed as an argument, the default file named **core** in the current directory is used.

Return Values

Upon successful completion, the **coredump** subroutine returns a value of 0. If the **coredump** subroutine is not successful, a value of -1 is returned and the **errno** global variable is set to indicate the error

Error Codes

Item	Description
EINVAL	Invalid argument.
EACCES	Search permission is denied on a component of the path prefix, the file exists and the pwrite permission is denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
EINPROGRESS	A core dump is already in progress.
ENOMEM	Not enough memory.
ENOTSUP	Routine not supported.
EFAULT	Invalid user address.

cosf, cosl, cos, cosd32, cosd64, and cosd128 Subroutines

Purpose

Computes the cosine.

Syntax

```
#include <math.h>

float cosf (x)
float x;

long double cosl (x)
long double x;

double cos (x)
double x;
_Decimal32 cosd32 (x)
_Decimal32 x;

_Decimal64 cosd64 (x)
_Decimal64 x;

_Decimal128 cosd128 (x)
_Decimal128 x;
```

Description

The **cosf**, **cosl**, **cos**, **cosd32**, **cosd64**, and **cosd218** subroutines compute the cosine of the *x*, parameter (measured in radians).

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **cosf**, **cosl**, **cos**, **cosd32**, **cosd64**, and **cosd128** subroutines return the cosine of *x*.

If *x* is NaN, a NaN is returned.

If *x* is ±0, the value 1.0 is returned.

If *x* is ±Inf, a domain error occurs, and a NaN is returned.

cosh, coshf, coshl, coshd32, coshd64, and coshd128 Subroutines

Purpose

Computes the hyperbolic cosine.

Syntax

```
#include <math.h>

float coshf (x)
float x;

long double coshl (x)
long double x;

double cosh (x)
double x;
_Decimal32 coshd32 (x)
_Decimal32 x;

_Decimal64 coshd64 (x)
_Decimal64 x;

_Decimal128 coshd128 (x)
_Decimal128 x;
```

Description

The **coshf**, **coshl**, **cosh**, **coshd32**, **coshd64**, and **coshd128** subroutines compute the hyperbolic cosine of the *x* parameter.

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. On return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **coshf**, **coshl**, **cosh**, **coshd32**, **coshd64**, and **coshd128** subroutines return the hyperbolic cosine of *x*.

If the correct value would cause overflow, a range error occurs and the **coshf**, **coshl**, **cosh**, **coshd32**, **coshd64**, and **coshd128** subroutines return the value of the macro **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±0, the value 1.0 is returned.

If *x* is ±Inf, +Inf is returned.

cpfile Subroutine

Purpose

Optimized copy operation of contents from the source file to the destination file.

Library

Standard C Library (`libc.a`)

Syntax

```
#include <unistd.h>

int cpfile(sfd, dfd, offset, nbytesp, flags)
int        sfd;
int        dfd;
off64_t    offset;
size64_t  *nbytesp;
uint64_t   flags;
```

Description

The **cpfile** subroutine copies `nbytes` data from the opened source file (ID specified in the *sfd* parameter) to the opened destination file (ID specified by the *dfd* parameter). The **cpfile** subroutine copies only regular files. The **cpfile** subroutine can copy files from one local file system, network file system (NFS) or, mounted file system to other file systems. If this function is used for any other type of file or file system, an error is returned.

The `offset` argument specifies where to begin the read operation from the source file and it starts writing to the same `offset` value in the destination file unless the destination file is opened in the append mode (by using **O_APPEND** flag). If the `offset` value is negative or indicates a position that is beyond the end of source file, an error is returned.

The *nbytesp* argument is an input and output argument. Basically this argument is used to pass the value also to return a value. For an input operation the address points to the number of bytes to be copied from the specified `offset` value. The value 0 copies the entire file (or until end of file, if the offset value is nonzero). Fewer bytes might be copied than requested because of the following reasons:

- Insufficient space to write in the destination file.
- Insufficient memory to allocate temporary buffers.
- Upper limit is reached or a pending signal is detected.

The error value of -1 is returned and the `errno` global variable is set to indicate the failure of copy operation. On return, the *nbytesp* value specifies how many bytes are successfully copied before the subroutine returned from the call.

If the **cpfile** subroutine is interrupted by any signal, it returns from kernel space to user space to handle the signal. The error number points to the EINTR value and the *nbytesp* value indicates the number of bytes copied before the **cpfile** subroutine was interrupted. If you want the application to continue copying bytes of data from the source file after the signal is handled, call the **cpfile** subroutine again with a new offset value and length.

Note: If the application restarts the operation where it stopped, it might need to specify the NO_DEST_FSIZE_CHECK flag because the destination file size might not be zero after the application returns from the first call.

The **flags** argument is used to control the behavior of the call to **cpfile** subroutine. Specify the value as 0 provided, if you do not want to use the flag. Any other value indicates a valid flag. Multiple flag values can be passed together as bits.

Supported options for flag values are as follows:

SPARSE_DEST_FILE
Source file blocks that have all zero strings are set by using the `fclear` operation instead of the write operation on the destination file. If source file is sparse then destination file also becomes sparse after the copy operation.

NO_DEST_FSIZE_CHECK
The `cpfile` subroutine must not check the size of any destination file. If this flag is set the **cpfile** subroutine overwrites the contents of destination file.

Consider the following information about the **cpfile** subroutine:

- The **cpfile** subroutine is used for one of the following purposes:
  - The **cpfile** subroutine is used to create identical copy of the source file. Destination file size must be zero, less than, or equal to source file size.

    Note: If destination file size is non-zero then **NO_DEST_FSIZE_CHECK** flag must be turned on. The **cpfile** subroutine does not explicitly truncate the destination file. Therefore the application must truncate the destination file to zero, less than, or equal to the source file size before the call. A non-zero sized destination file that is opened in append mode does not create an identical copy of the source file after the copy operation is complete.
  - The **cpfile** subroutine is used to replace the portion of the destination file at the offset value that is specified by the `offset` parameter by the portion of the source file at the same offset value. Therefore, the destination file size can be nonzero. In this case, the application must turn on the **NO_DEST_FSIZE_CHECK** flag.
  - The **cpfile** subroutine is used to concatenate the content of source file with the content of destination file. In this case, the application must open destination file in append mode (by using the **O_APPEND** flag) and the **NO_DEST_FSIZE_CHECK** flag must be turned on because the destination file size is nonzero. The **cpfile** subroutine starts appending data from the source file to the end of the destination file.
- The **cpfile** subroutine can be used in a multi-threaded environment.
- When the subroutine is copying data, a parallel write operation on the source file, or the destination file might result in an unexpected result.
- If the **SPARSE_DEST_FILE** flag is specified, the **cpfile** subroutine optimizes the copy operation of the source file by skipping the block that has all zero strings in the source file by using the **fclear** flag instead of performing the write operation on the destination file.
- The **cpfile** subroutine does not copy any attributes, extended attributes, access control lists (ACLs) from the source file to the destination file. You must manually copy these attributes, if required.
- If system call detects any pending signal, the **cpfile** subroutine returns from kernel space to user space to handle the signal for an application. If application continues the copy operation after the pending signal is processed, application must call the **cpfile** subroutine again with a new offset value and length. New offset value indicates that from where to continue the copy operation and new length indicates the bytes to be copied from the specified offset.

  Note: The application might need to specify the **NO_DEST_FSIZE_CHECK** flag for consecutive calls because the destination file size might be non-zero after the previous call to the **cpfile** subroutine.
- By default, the **cpfile** subroutine expects the destination file size to be zero. If the application wants to work with the destination file of size non-zero, the application must pass the **NO_DEST_FSIZE_CHECK** flag to avoid failure. If application wants to concatenate the source file with the destination file, it must open the destination file in append mode (by using the**O_APPEND** flag) and call the **cpfile** subroutine with the **NO_DEST_FSIZE_CHECK** flag turned on.

  Note: If the application specifies the **NO_DEST_FSIZE_CHECK** flag, the destination file size is not checked. Hence, if the destination file is larger than source file, data in the destination file, which is located after an offset equal to the source file size is not modified by the **cpfile** subroutine.

Parameters

sfd
Specifies the file descriptor for the source file.

dfd
Specifies the file descriptor for the destination file.

Offset
Specifies the position in the source file from where to read the data and the position in the destination file where to start writing the data.

nbytesp
Specifies the input value or output value. This argument specifies the number of bytes to be copied. The value 0 copies the entire file. This argument returns the number of bytes that are copied.

flags
Specifies flag values as defined by parameters of the subroutine in the description section.

Return values

Upon successful completion, the call to the **cpfile** subroutine returns 0. The number of bytes that is copied to the destination file is must not be greater than the value specified by the *nbytes* parameter. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate an error. In both the cases, the *nbytesp* variable has the value of number of bytes that was copied to the destination file.

Error codes

The **cpfile** subroutine is unsuccessful when one or more of the following error codes are true. File system can generate errors other than the errors specified in the following list:

EBADF
The file descriptor parameter is not valid.

EINTR
The operation was interrupted by a signal.

EINVAL
The offset, length, or flags parameter is invalid or the *nbytesp* parameter is null. If destination file size is nonzero and if the **NO_DEST_FSIZE_CHECK** flag is not set, the **EINVAL** error code is returned.

ENOMEM
No memory is available in the system to perform the I/O operation.

EFBIG
An offset value greater than the **MAX_FILESIZE** value was requested.

EAGAIN
The source or destination file was changed unexpectedly. Error code indicates that the `cpfile` subroutine must be called again.

Example

The following code fragment shows the optimized method to copy file by using the **cpfile** subroutine:

```
#include <unistd.h>
int main (int argc, char **argv)
{
       int sfd, dfd;
       size64_t  nbytes = 0;
       uint64_t  flags = 0;
       off64_t   offset = 0;

       /* Open source file */
       sfd = open(argv[0], O_RDONLY, 0);
       if (sfd < 0)
       {
             perror("open");
```

```
                exit(-1);
        }
        /* Open destination file. Create if not exist and truncate to zero size. */
        dfd = open(argv[1],O_RDWR|O_CREAT|O_TRUNC, 0644);
        if (dfd < 0)
        {
                perror("open");
                exit(-1);
        }

        /* Perform any other tasks like copying attributes */
        /* Call cpfile to copy whole file. */
        rc = cpfile(sfd, dfd, offset, &nbytes, flags);
        {
                perror("cpfile");
                exit(-1);
        }
        close(sfd);
        close(dfd);
}
```

cpow, cpowf, or cpowl Subroutine

Purpose

Computes the complex power.

Syntax

```
#include <complex.h>

double complex cpow (x, y)
double complex x;
double complex y;

float complex cpowf (x, y)
float complex x;
float complex y;

long double complex cpowl (x, y)
long double complex x;
long double complex y;
```

Description

The **cpow**, **cpowf**, and **cpowl** subroutines compute the complex power function x^y, with a branch cut for the first parameter along the negative real axis.

Parameters

Item	Description
x	Specifies the base value.
y	Specifies the power the base value is raised to.

Return Values

The **cpow**, **cpowf**, and **cpowl** subroutines return the complex power function value.

cproj, cprojf, or cprojl Subroutine

Purpose

Computes the complex projection functions.

Syntax

```
#include <complex.h>

double complex cproj (z)
double complex z;

float complex cprojf (z)
float complex z;

long double complex cprojl (z)
long double complex z;
```

Description

The **cproj**, **cprojf**, and **cprojl** subroutines compute a projection of *z* onto the Riemann sphere: *z* projects to *z*, except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If *z* has an infinite part, **cproj**(*z*) shall be equivalent to:

```
INFINITY + I * copysign(0.0, cimag(z))
```

Parameters

Item	Description
z	Specifies the value to be projected.

Return Values

The **cproj**, **cprojf**, and **cprojl** subroutines return the value of the projection onto the Riemann sphere.

cpu_context_barrier and cpu_speculation_barrier Subroutines

Purpose

Provides protection against speculative execution side-channel attacks.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/processor.h>
```

void cpu_context_barrier (int *value*)

void cpu_speculation_barrier (*void*)

Description

The **cpu_context_barrier** and **cpu_speculation_barrier** subroutines provide applications with processor-model-dependent mitigation against known speculative-execution vulnerabilities. These subroutines can be used by both 32-bit and 64-bit applications to protect applications against data-dependent storage access and to provide isolation between the trusted and untrusted segments of an application.

Note: Application performance might reduce when the **cpu_context_barrier** or **cpu_speculation_barrier** subroutine is used.

The **cpu_context_barrier** subroutine must be called from within the trusted domain and must be executed at each transition between the trusted domain and the untrusted domain. This subroutine accepts a single parameter that specifies the method in which the subroutine is used. Alternatively, a comprehensive variation of the barrier kernel subroutine can be used for scenarios where it is difficult to distinguish the method in which the subroutine must be used.

The **cpu_speculation_barrier** subroutine must be called from within the trusted domain before storage is accessed by using addresses that are computed from an untrusted source.

Parameters

Item	Description
value	Specifies the method in which the barrier subroutine is being invoked.

CPU context barrier values

Item	Description
CCB_ENTRY	Specify this value when transitioning into a trusted context domain.
CCB_EXIT	Specify this value when transitioning out of a trusted context domain.
CCB_ALL	Specify this value when transitioning into a trusted context domain or transitioning out of a trusted context domain.

Example

The following example shows how the trusted domain of an application calls an untrusted domain:

```
int                index;
char               val,
                   udata[];
extern int         max_tdata_index;
extern char        tdata[];

/* Fetch index from untrusted user */
cpu_context_barrier(CCB_EXIT);
index = get_index_from_user(...);
cpu_context_barrier(CCB_ENTRY);

/* Select trusted data from user input */
if (index < max_tdata_index) {
        cpu_speculation_barrier();
        val = tdata[index];
        udata[val]++;
}
```

cpuextintr_ctl Subroutine

Purpose

Performs Central Processing Unit (CPU) external interrupt control related operations on CPUs.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/intr.h>

int cpuextintr_ctl(command,cpuset,flags)
```

```
extintrctl_t command;
rsethandle_t cpuset;
uint flags;
```

Description

The **cpuextintr_ctl** subroutine provides means of enabling, disabling, and querying the external interrupt state on the CPUs described by the CPU resource set. If you enable or disable a CPU's external interrupt, it affects the external interrupt delivery to the CPU. Typically, on multiple CPU system, external interrupts can be delivered to any running CPU, and the distribution among the CPUs is determined by a predefined method. Any external interrupt can only be delivered to a CPU if its interrupt priority is more favored than the current external interrupt priority of the CPU. When external interrupts are disabled through this interface, any external interrupt priority that is less favored than INTMAX is blocked until interrupts are enabled again. The **cpuextintr_ctl** subroutine is applicable only on selective hardware types.

Note: Because this subroutine changes the way external interrupt is delivered, system performance can be affected. This service guarantees at least one online CPU is available to handle all the external interrupts. Any CPU DLPAR removal fails if the operation breaks such rule. On an I/O bound system, one CPU might not be enough to handle all the external interrupts. Performance suffers due to insufficient CPU available to handle external interrupts.

Parameters

Item	Description
command	Specifies the operation to the CPUs specified by CPU resource set. One of the following values that are defined in **<sys/intr.h>** file can be used: **EXTINTDISABLE** Disable external interrupt on the CPUs specified by the CPU resource set. **EXTINTENABLE** Enable external interrupt on the CPUs specified by the CPU resource set. **QUERYEXTINTDISABLE** Returns a CPU resource set that have the CPUs with external interrupt as disabled. **QUERYEXTINTENABLE** Returns a CPU resource set that have the CPUs with external interrupt as enabled.
cpuset	Reference to a CPU resource set. Upon successful return from this kernel service, the CPUs, for which the external interrupt control operation is complete are set in the CPU resource set. The CPUs specified by the **cpuset** parameter are logical CPU IDs.
flags	Always set to 0 or EINVAL is returned.

Security

The caller must have root authority with the **CAP_NUMA_ATTACH** capability or **PV_KER_CONF** privilege in the RBAC environment.

Return Values

Upon successful completion, the **cpuextintr_ctl** subroutine returns the number of CPUs on which the command successfully completed. If unsuccessful, -1 is returned and the errno global variable is set to indicate the error.

Error Codes

Item	Description
EINVAL	The command is not valid, the **cpuset** references NULL, the **cpuset** is empty, or the flags value is unknown.
EFAULT	The **cpuset** buffer passed in is not valid.
ENOSYS	This function is not implemented on the platform.
EPERM	Caller does not have enough privilege to perform the requested operation.

Note: A return value of success does not necessarily indicate that external interrupts have been enabled or disabled on all of the specified CPUs. For example, if a CPU is not online, the enable or disable operation will not be performed on that CPU. The caller must check the returned cpuset to verify the completion of this operation on the CPUs. The **k_cpuextintr_ctl** kernel service does not block DR CPU add or remove operation during the entire period of system call.

creal, crealf, or creall Subroutine

Purpose

Computes the real part of a specified value.

Syntax

```
#include <complex.h>

double creal (z)
double complex z;

float crealf (z)
float complex z;

long double creall (z)
long double complex z;
```

Description

The **creal**, **crealf**, and **creall** subroutines compute the real part of the value specified by the *z* parameter.

Parameters

Item	Description
z	Specifies the real to be computed.

Return Values

These subroutines return the real part value.

crypt, encrypt, or setkey Subroutine

Purpose

Encrypts or decrypts data.

Library

Standard C Library (**libc.a**)

Syntax

```
char *crypt (PW, Salt)
const char * PW, * Salt;

void encrypt (Block, EdFlag)
char Block[64];
int EdFlag;

void setkey (Key)
const char * Key;
```

Description

The **crypt** and **encrypt** subroutines encrypt or decrypt data. The **crypt** subroutine performs a one-way encryption of a fixed data array with the supplied *PW* parameter. The subroutine uses the *Salt* parameter to vary the encryption algorithm.

The **encrypt** subroutine encrypts or decrypts the data supplied in the *Block* parameter using the key supplied by an earlier call to the **setkey** subroutine. The data in the *Block* parameter on input must be an array of 64 characters. Each character must be an char 0 or char 1.

If you need to statically bind functions from **libc.a** for **crypt** do the following:

1. Create a file and add the following:

   ```
   #!
   ___setkey
   ___encrypt
   ___crypt
   ```

2. Perform the linking.
3. Add the following to the make file:

   ```
   -bI:YourFileName
   ```

   where *YourFileName* is the name of the file you created in step 1. It should look like the following:

   ```
   LDFLAGS=bnoautoimp -bI:/lib/syscalls.exp -bI:YourFileName -lc
   ```

These subroutines are provided for compatibility with UNIX system implementations.

Parameters

Item	Description
Block	Identifies a 64-character array containing the values (**char**) 0 and (**char**) 1. Upon return, this buffer contains the encrypted or decrypted data.
EdFlag	Determines whether the subroutine encrypts or decrypts the data. If this parameter is 0, the data is encrypted. If this parameter is a nonzero value, the data is decrypted. If the **/usr/lib/libdes** or **/usr/lib/libdes_64** file does not exist and if the *EdFlag* parameter is set to a nonzero value, the **encrypt** subroutine returns the **ENOSYS** error code. The **/usr/lib/libdes** and **/usr/lib/libdes_64** files are part of the **des** fileset, which is located in the AIX Expansion Pack.
Key	Specifies an 64-element array of 0's and 1's cast as a **const char** data type. The *Key* parameter is used to encrypt or decrypt data.

Item	Description
PW	Specifies the string to be encrypted.
Salt	Determines the algorithm that the *PW* parameter applies to generate the returned output string. If the left brace ({) is not the first character of the value that the *Salt* parameter specifies, then the subroutine uses the Data Encryption Standard (DES) algorithm. For the DES algorithm, use the *Salt* parameter to vary the hashing algorithm in the one of 4096 ways. The *Salt* parameter must be a 2-character string that is from the following character types: **A-Z** Uppercase alpha characters **a-z** Lowercase alpha characters **0-9** Numeric characters **.** Period **/** Slash If the left brace ({) is the first character of the value that the *Salt* parameter specifies, then the Loadable Password Algorithm (LPA) uses the name that is specified within the braces ({}). A set of salt characters follows the LPA name and ends with a dollar sign ($). The length of the salt character depends on the specified LPA. The following example shows a possible value for the SMD5 LPA that the *Salt* parameter specifies: `{SMD5}JVDbGx8K$`

Return Values

The **crypt** subroutine returns a pointer to the encrypted password. The static area this pointer indicates may be overwritten by subsequent calls.

If the **crypt** subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **encrypt** subroutine returns the following error codes:

Item	Description
ENOSYS	The **encrypt** subroutine was called by using the *EdFlag* parameter that was set to a nonzero value. Also, the **/usr/lib/libdes** or **/usr/lib/libdes_64** file does not exist. The **/usr/lib/libdes** and **/usr/lib/libdes_64** files are part of the **des** fileset, which is located in the AIX Expansion Pack.

csid Subroutine

Purpose

Returns the character set ID (charsetID) of a multibyte character.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

int csid ( String)
const char *String;
```

Description

The **csid** subroutine returns the charsetID of the multibyte character pointed to by the *String* parameter. No validation of the character is performed. The parameter must point to a value in the character range of the current code set defined in the current locale.

Parameters

Item	Description
String	Specifies the character to be tested.

Return Values

Successful completion returns an integer value representing the charsetID of the character. This integer can be a number from 0 through *n*, where *n* is the maximum character set defined in the CHARSETID field of the **charmap**.

csin, csinf, or csinl Subroutine

Purpose

Computes the complex sine.

Syntax

```
#include <complex.h>

double complex csin (z)
double complex z;

float complex csinf (z)
float complex z;

long double complex csinl (z)
long double complex z;
```

Description

The **csin**, **csinf**, and **csinl** subroutines compute the complex sine of the value specified by the *z* parameter.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **csin**, **csinf**, and **csinl** subroutines return the complex sine value.

csinh, csinhf, or csinhl Subroutine

Purpose

Computes the complex hyperbolic sine.

Syntax

```
#include <complex.h>

double complex csinh (z)
double complex z;

float complex csinhf (z)
float complex z;

long double complex csinhl (z)
long double complex z;
```

Description

The **csinh**, **csinhf**, and **csinhl** subroutines compute the complex hyperbolic sine of the value specified by the *z* parameter.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **csinh**, **csinhf**, and **csinhl** subroutines return the complex hyperbolic sine value.

csqrt, csqrtf, or csqrtl Subroutine

Purpose

Computes complex square roots.

Syntax

```
#include <complex.h>
double complex csqrt (z)
double complex z;

float complex csqrtf (z)
float complex z;

long double complex csqrtl (z)
long double complex z;
```

Description

The **csqrt**, **csqrtf**, and **csqrtl** subroutines compute the complex square root of the value specified by the *z* parameter, with a branch cut along the negative real axis.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **csqrt**, **csqrtf**, and **csqrtl** subroutines return the complex square root value, in the range of the right half-plane (including the imaginary axis).

CT_HOOKx and CT_GEN macros

Purpose

Record a trace event into Component Trace, LMT or system trace buffers.

Syntax

The following set of macros is provided to record a trace entry:

```
#include <sys/ras_trace.h>
CT_HOOK0(ras_block_t cb, int level, int mem_dest,long hkwd);
CT_HOOK1(ras_block_t cb, int level, int mem_dest, long hkwd, long d1);
CT_HOOK2(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2);
CT_HOOK3(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2, long d3);
CT_HOOK4(ras_block_t cb, int level, \
int mem_dest, long hkwd, long d1, long d2, \
long d3, long d4);
CT_HOOK5(ras_block_t cb, int level, int mem_dest, \
long hkwd, long d1, long d2, long d3, \
long d4, long d5);
CT_GEN (ras_block_t cb, int level, long hkwd, long data, long len, void *buf);
```

Description

The CT_HOOKx macros allow you to record a trace hook. The "x" is the number of data words you want in this trace event.

The CT_GEN macro is used to record a generic trace hook.

All traces are timestamped.

Restriction: If the *cb* input parameter has a value of RAS_BLOCK_NULL, no tracing will be performed.

Parameters

Item	Description
`ras_block_t` *cb*	The *cb* parameter in the RAS control block that refers to the component that this trace entry belongs to.

Item	Description
`int` *level*	The *level* parameter allows filtering of different trace entries. The higher this level is, the more this trace will be considered as debug or detail information. In other words, this trace entry will appear only if the level of the trace entry is less than or equal to the level of trace chosen for memory or system trace mode. Ten levels of trace are available (**CT_LEVEL_0** to **CT_LEVEL_9**, corresponding to value 0 to 9) with four special levels: • minimal (CT_LVL_MINIMAL (=CT_LEVEL_1)) • normal (CT_LVL_NORMAL (=CT_LEVEL_3)) • detail (CT_LVL_DETAIL (=CT_LEVEL_7)) • default (CT_LVL_DEFAULT = CT_LVL_NORMAL in AIX 6.1 and above and CT_LVL_MINIMAL otherwise) When you are porting an existing driver or subsystem from the existing system trace to component trace, trace existing entries at **CT_LVL_DEFAULT**.
`int` *mem_dest*	For `CT_HOOKx` macros, the *mem_dest* parameter indicates the memory destination for this trace entry. It is an ORed value with the following possible settings: • `MT_RARE`: the trace entry is saved in the rare buffer of lightweight memory trace if the level condition of the memory trace mode for this control block is satisfied, meaning that the current level of trace for the memory trace mode is greater than or equal to the level of this trace entry. • `MT_COMMON`: the trace entry is saved in the common buffer of the lightweight memory trace if the level condition of the memory trace mode for this control block is satisfied. • `MT_PRIV`: the trace entry is saved in the private memory buffer of the component if the level condition of the memory trace mode for this control block is satisfied. • `MT_SYSTEM`: the trace entry is saved in the existing system trace if the level condition of the *system trace mode* for this control block is satisfied, if the system trace is running, and if the hook meets any additional criteria specified as part of the system trace. For example, if `MT_SYSTEM` is not set, the trace entry is not saved in the existing system trace. Only one of the `MT_RARE`, `MT_COMMON` and `MT_PRIV` values should be used, but you can combine ORed with `MT_SYSTEM`. Otherwise, the trace entry will be duplicated in several memory buffers. The *mem_dest* parameter is not needed for the `CT_GEN` macro because lightweight memory trace cannot accommodate generic entries. `CT_GEN` checks the memory trace and system trace levels to determine whether the generic entry should enter the private memory buffer and system trace buffers respectively.

The *hkwd*, *d1*, *d2*, *d3*, *d4*, *d5*, *len* and *buf* parameters are the same as those used for the existing TRCHKx or TRCGEN macros. The TRCHKx refers to the TRCHKLnT macros where *n* is from 0 to 5. For example, TRCHKL1T (hkwd, d1). The TRCGEN macros refer to the TRCGEN and TRCGENT macros.

For the hookword, OR the hookID with a subhookID if needed. For the `CT_HOOKx` macro, the subhook is ORed into the hookword. For the `CT_GEN` macro, the subhook is the *d1* parameter.

CT_HOOKx_PRIV, CTCS_HOOKx_PRIV, CT_HOOKx_COMMON, CT_HOOKx_RARE, and CT_HOOKx_SYSTEM Macros

Purpose

Record a trace event into Component Trace (CT), Lightweight Memory Trace (LMT), or system trace buffers.

Syntax

```
#include <sys/ras_trace.h>
CT_HOOK0_PRIV(ras_block_t cb, ulong hw);
CT_HOOK1_PRIV(ras_block_t cb, ulong hw, ulong d1);
CT_HOOK2_PRIV(ras_block_t cb, ulong hw, ulong d1, ulong d2);
CT_HOOK3_PRIV(ras_block_t cb, ulong hw, ulong d1, ulong d2, ulong d3);
CT_HOOK4_PRIV(ras_block_t cb, ulong hw, ulong d1, ulong d2, ulong d3, ulong d4);
CT_HOOK5_PRIV(ras_block_t cb, ulong hw, ulong d1, ulong d2, ulong d3, ulong d4, ulong d5);
```

```
#include <sys/ras_trace.h>
CTCS_HOOK0_PRIV(ras_block_t cb, ulong hw);
CTCS_HOOK1_PRIV(ras_block_t cb, ulong hw, ulong d1);
CTCS_HOOK2_PRIV(ras_block_t cb, ulong hw, ulong d1, ulong d2);
CTCS_HOOK3_PRIV(ras_block_t cb, ulong hw, ulong d1, ulong d2, ulong d3);
CTCS_HOOK4_PRIV(ras_block_t cb, ulong hw, ulong d1, ulong d2, ulong d3, ulong d4);
CTCS_HOOK5_PRIV(ras_block_t cb, ulong hw, ulong d1, ulong d2, ulong d3, ulong d4, ulong d5);
```

```
#include <sys/ras_trace.h>
CT_HOOK0_COMMON(ulong hw);
CT_HOOK1_COMMON(ulong hw, ulong d1);
CT_HOOK2_COMMON(ulong hw, ulong d1, ulong d2);
CT_HOOK3_COMMON(ulong hw, ulong d1, ulong d2, ulong d3);
CT_HOOK4_COMMON(ulong hw, ulong d1, ulong d2, ulong d3, ulong d4);
CT_HOOK5_COMMON(ulong hw, ulong d1, ulong d2, ulong d3, ulong d4, ulong d5);
```

```
#include <sys/ras_trace.h>
CT_HOOK0_RARE(ulong hw);
CT_HOOK1_RARE(ulong hw, ulong d1);
CT_HOOK2_RARE(ulong hw, ulong d1, ulong d2);
CT_HOOK3_RARE(ulong hw, ulong d1, ulong d2, ulong d3);
CT_HOOK4_RARE(ulong hw, ulong d1, ulong d2, ulong d3, ulong d4);
CT_HOOK5_RARE(ulong hw, ulong d1, ulong d2, ulong d3, ulong d4, ulong d5);
```

```
#include <sys/ras_trace.h>
CT_HOOK0_SYSTEM(ulong hw);
CT_HOOK1_SYSTEM(ulong hw, ulong d1);
CT_HOOK2_SYSTEM(ulong hw, ulong d1, ulong d2);
CT_HOOK3_SYSTEM(ulong hw, ulong d1, ulong d2, ulong d3);
CT_HOOK4_SYSTEM(ulong hw, ulong d1, ulong d2, ulong d3, ulong d4);
CT_HOOK5_SYSTEM(ulong hw, ulong d1, ulong d2, ulong d3, ulong d4, ulong d5);
```

Description

The CT_HOOKx_PRIV, CTCS_HOOKx_PRIV, CT_HOOKx_COMMON, CT_HOOKx_RARE, and CT_HOOKx_SYSTEM macros trace a trace event in to a specific trace facility. These macros are optimized for performance. Due to this optimization, no explicit checking is done to ensure the availability of a trace facility. In general, it is always safe to trace to either of the LMT buffer types or system source. Callers should use the **rasrb_trace_privlevel()** service to ensure that the selected Component Trace private buffer is available. Before calling routines that write to the private buffer of a Component Trace, checks should be made to ensure that the return value is not -1, and that the buffer is at the appropriate level required for tracing. Race conditions for infrastructure-serialized Component Trace macros are handled by the infrastructure. Component-serialized traces must ensure proper serialization between tracing and state changes made in the corresponding RAS callback.

The following table describes how macros are associated with a specific trace facility and includes notes about the macros.

Item	**Description**	
Trace Facility	**Macro**	**Notes**
Component Trace private buffer	CT_HOOKx_PRIV	Can be used with both infrastructure and component serialized traces.
Component Trace private buffer	CTCS_HOOKx_PRIV	Can only be used with component serialized traces.
Lightweight Memory Trace common buffer	CT_HOOKx_COMMON	
Lightweight Memory Trace rare buffer	CT_HOOKx_RARE	
System Trace buffer	CT_HOOKx_SYSTEM	

All traces are recorded with time stamps.

If the *cb* input parameter has a value of RAS_BLOCK_NULL, no tracing is performed.

Parameters

Item	**Description**
ras_block_t *cb*	The *cb* parameter is the RAS control block that refers to the component that this trace entry belongs to.

The *hkwd*, *d1*, *d2*, *d3*, *d4*, and *d5* parameters are the same as those used for the existing TRCHKx macros. The TRCHKx refers to the TRCHKLnT macros where *n* is from 0 to 5. For example, TRCHKL1T (hkwd, d1).

Example

In the following example, the **foo()** function uses Component Trace private buffers with system trace in a performance optimized way. The **foo()** function uses component-serialization and traces only when the detail level is at or above the CT_LEVEL_NORMAL level (defined in **sys/ras_trace.h**).

```
void foo() {
    long ipl;
    char memtrc, systrc;

    ipl = disable_lock(INTMAX, <Component Trace lock>);
    memtrc = rasrb_trace_privlevel(rasb) >= CT_LVL_NORMAL ? 1 : 0;
    systrc = rasrb_trace_syslevel(rasb) >= CT_LVL_NORMAL ? 1 : 0;
    …
    if (memtrc) {
        CTCS_HOOK5_PRIV(…)
    }
    if (systrc) {
        __INFREQUENT();
        CT_HOOK5_SYSTEM(…)
    }
    …
    unlock_enable(ipl, <Component Trace lock>)
    return;
}
```

CT_TRCON macro

Purpose

Return information on whether any trace is active at a certain level for a component.

Syntax

```
#include <sys/ras_trace.h>
CT_TRCON(cb, level)
```

Description

The CT_TRCON macro allows you to ascertain whether any type of trace (Component Trace, lightweight memory trace or system trace) will record events for the component specified at the trace detail level specified.

Note: If the *cb* input parameter has a value of RAS_BLOCK_NULL, the **CT_TRCON** macro indicates that the trace is off.

Parameters

Item	Description
`ras_block_t` *cb*	The *cb* parameter is the RAS control block pointer that refers to the component that this trace entry belongs to.
`int` *level*	Specifies the trace detail level.

ctan, ctanf, or ctanl Subroutine

Purpose

Computes complex tangents.

Syntax

```
#include <complex.h>

double complex ctan (z)
double complex z;

float complex ctanf (z)
float complex z;

long double complex ctanl (z)
long double complex z;
```

Description

The **ctan**, **ctanf**, and **ctanl** subroutines compute the complex tangent of the value specified by the *z* parameter.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **ctan**, **ctanf**, and **ctanl** subroutines return the complex tangent value.

ctanh, ctanhf, or ctanhl Subroutine

Purpose

Computes the complex hyperbolic tangent.

Syntax

```
#include <complex.h>

double complex ctanh (z)
double complex z;

float complex ctanhf (z)
float complex z;

long double complex ctanhl (z)
long double complex z;
```

Description

The **ctanh**, **ctanhf**, and **ctanhl** subroutines compute the complex hyperbolic tangent of *z*.

Parameters

Item	Description
z	Specifies the value to be computed.

Return Values

The **ctanh**, **ctanhf**, and **ctanhl** subroutines return the complex hyperbolic tangent value.

CTCS_HOOKx Macros

Purpose

Record a trace event into component serialized Component Trace, Lightweight Memory Trace (LMT), or system trace buffers.

Syntax

The following set of macros is provided to record a trace entry:

```
#include <sys/ras_trace.h>
CTCS_HOOK0(ras_block_t cb, int level, int mem_dest, long hkwd);
CTCS_HOOK1(ras_block_t cb, int level, int mem_dest, long hkwd, long d1);
CTCS_HOOK2(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2);
CTCS_HOOK3(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2, long d3);
CTCS_HOOK4(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2, long d3, long
d4);
CTCS_HOOK5(ras_block_t cb, int level, int mem_dest, long hkwd, long d1, long d2, long d3, long
d4,
long d5);
```

Description

The **CTCS_HOOKx** macros record a trace hook in to a Component Trace buffer that is component-serialized. These macros cannot be used with buffers that are not component-serialized. The **x** in **CTCS_HOOKx** is the number of data words you want in this trace event.

All of the traces that are recorded are time-stamped.

If the *cb* input parameter contains a value of RAS_BLOCK_NULL, no tracing is performed.

Parameters

Item	Description
ras_block_t *cb*	The *cb* parameter is the RAS control block that links to the component that this trace entry belongs to.
int *level*	The *level* parameter allows filtering of different trace entries. The higher this level is, the more this trace is considered as debug or detail information. This trace entry is displayed only if the level of the trace entry is less than or equal to the level of the trace chosen for memory or system trace mode. Ten levels of trace are available (CT_LEVEL_0 to CT_LEVEL_9, corresponding to value 0 to 9) with the following special levels: • Minimal (CT_LVL_MINIMAL (=CT_LEVEL_1)) • Normal (CT_LVL_NORMAL (=CT_LEVEL_3)) • Detail (CT_LVL_DETAIL (=CT_LEVEL_7)) • Default (CT_LVL_DEFAULT = CT_LVL_NORMAL in AIX 6.1 and above. Otherwise, it is CT_LVL_MINIMAL) When you are porting an existing driver or subsystem from the existing system trace to a component trace, existing entries should be traced at the CT_LVL_DEFAULT level.

Item	Description
int *mem_dest*	The *mem_dest* parameter indicates the memory destination for this trace entry. It is an ORed value with the following possible settings: **MT_RARE** The trace entry is saved in the rare buffer of lightweight memory. In this case, the current level of trace for the memory trace mode is greater than or equal to the level of this trace entry. **MT_COMMON** The trace entry is saved in the common buffer of the lightweight memory trace. **MT_PRIV** The trace entry is saved in the private memory buffer of the component. **MT_SYSTEM** The trace entry is saved in the existing system trace if all of the following conditions are true: • The level condition of the system trace mode for this control block is satisfied • The system trace is running • The hook meets any additional criteria specified as part of the system trace If MT_SYSTEM is not set, the trace entry is not saved in the existing system trace. Only one of the **MT_RARE**, **MT_COMMON**, and **MT_PRIV** values should be used, but you can combine ORed with **MT_SYSTEM**. Otherwise, the trace entry will be duplicated in several memory buffers. The *mem_dest* parameter is not necessary for the **CT_GEN** macro because Lightweight Memory Trace cannot accommodate generic entries. The **CT_GEN** macro checks the memory trace and system trace levels to determine whether the generic entry should enter the private memory buffer and the system trace buffers respectively.

The *hkwd*, *d1*, *d2*, *d3*, *d4*, and *d5* parameters are the same as those used for the existing TRCHKx macros. The TRCHKx macros link to the TRCHKLnT macros where *n* is from 0 to 5. For example, TRCHKL1T (hkwd, d1).

ctermid Subroutine

Purpose

Generates the path name of the controlling terminal.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
char *ctermid ( String)
char *String;
```

Description

The **ctermid** subroutine generates the path name of the controlling terminal for the current process and stores it in a string.

Note: File access permissions depend on user access. Access to a file whose path name the **ctermid** subroutine has returned is not guaranteed.

The difference between the **ctermid** and **ttyname** subroutines is that the **ttyname** subroutine must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor. The **ctermid** subroutine returns a string (the **/dev/tty** file) that refers to the terminal if used as a file name. Thus, the **ttyname** subroutine is useful only if the process already has at least one file open to a terminal.

Parameters

Item	Description
String	If the *String* parameter is a null pointer, the string is stored in an internal static area and the address is returned. The next call to the **ctermid** subroutine overwrites the contents of the internal static area. If the *String* parameter is not a null pointer, it points to a character array of at least `L_ctermid` elements as defined in the **stdio.h** file. The path name is placed in this array and the value of the *String* parameter is returned.

CTFUNC_HOOKx Macros

Purpose

Record a trace event, which is infrequently recorded, into Component Trace (CT), Lightweight Memory Trace (LMT), or system trace buffers.

Syntax

```
#include <sys/ras_trace.h>
CTFUNC_HOOK0(ras_block_t cb, char level, int mem_dest, ulong hw);
CTFUNC_HOOK1(ras_block_t cb, char level, int mem_dest, ulong hw, ulong d1);
CTFUNC_HOOK2(ras_block_t cb, char level, int mem_dest, ulong hw, ulong d1, ulong d2);
CTFUNC_HOOK3(ras_block_t cb, char level, int mem_dest, ulong hw, ulong d1, ulong d2, ulong d3);
CTFUNC_HOOK4(ras_block_t cb, char level, int mem_dest, ulong hw, ulong d1, ulong d2, ulong d3, ulong d4);
CTFUNC_HOOK5(ras_block_t cb, char level, int mem_dest, ulong hw, ulong d1, ulong d2, ulong d3, ulong d4,
ulong d5);
```

Description

The **CTFUNC_HOOKx** macros record a trace hook. Theses macros are optimized to record events that are rarely recorded, such as error path tracing. The **CTFUNC_HOOKx** macros can be used with any types of trace serialization. Besides their optimization for rare events, the **CTFUNC_HOOKx** macros are equivalent to the **CT_HOOKx** macros.

All of the traces that the **CTFUNC_HOOKx** macros record are time-stamped.

If the *cb* input parameter contains a value of RAS_BLOCK_NULL, no tracing will be performed.

Parameters

Item	Description
ras_block_t *cb*	The *cb* parameter is the RAS control block that refers to the component that this trace entry belongs to.

Item	Description
char *level*	The *level* parameter allows filtering of different trace entries. The higher this level is, the more this trace is considered as debug or detail information. This trace entry appears only if the level of the trace entry is less than or equal to the level of trace chosen for memory or system trace mode. Ten levels of trace are available (CT_LEVEL_0 to CT_LEVEL_9, corresponding to value 0 to 9) with the following four special levels: • Minimal (CT_LVL_MINIMAL (=CT_LEVEL_1)) • Normal (CT_LVL_NORMAL (=CT_LEVEL_3)) • Detail (CT_LVL_DETAIL (=CT_LEVEL_7)) • Default (CT_LVL_DEFAULT = CT_LVL_NORMAL in AIX 6.1. Otherwise, it is CT_LVL_MINIMAL) When you are porting an existing driver or subsystem from the existing system trace to component trace, existing entries should be traced at CT_LVL_DEFAULT.
int *mem_dest*	The *mem_dest* parameter indicates the memory destination for this trace entry. It is an ORed value with the following possible settings: **MT_RARE** The trace entry is saved in the rare buffer of lightweight memory trace if the level condition of the memory trace mode for this control block is satisfied, which means the current level of trace for the memory trace mode is greater than or equal to the level of this trace entry. **MT_COMMON** The trace entry is saved in the common buffer of the lightweight memory trace if the level condition of the memory trace mode for this control block is satisfied. **MT_PRIV** The trace entry is saved in the private memory buffer of the component if the level condition of the memory trace mode for this control block is satisfied. **MT_SYSTEM** The trace entry is saved in the existing system trace if all of the following conditions are true: • The level condition of the system trace mode for this control block is satisfied. • The system trace is running. • The hook meets any additional criteria specified as part of the system trace. If MT_SYSTEM is not set, the trace entry is not saved in the existing system trace. Only one of the **MT_RARE**, **MT_COMMON**, and **MT_PRIV** values can be used, but you can combine ORed with **MT_SYSTEM**. Otherwise, the trace entry duplicates in several memory buffers. The *mem_dest* parameter is not necessary for the **CT_GEN** macro because lightweight memory trace cannot accommodate generic entries. The **CT_GEN** macro checks the memory trace and system trace levels to determine whether the generic entry should enter the private memory buffer and the system trace buffers respectively.

The *hkwd*, *d1*, *d2*, *d3*, *d4*, and *d5* parameters are the same as those used for the existing TRCHKx macros. The TRCHKx macros link to the TRCHKLnT macros where *n* is from 0 to 5. For example, TRCHKL1T (hkwd, d1).

ctime, localtime, gmtime, mktime, difftime, asctime, or tzset Subroutine

Purpose

Converts the formats of date and time representations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>
```

```
char *ctime ( Clock)
const time_t *Clock;
```

```
struct tm *localtime (Clock)
const time_t *Clock;
```

```
struct tm *gmtime (Clock)
const time_t *Clock;
```

```
time_t mktime( Timeptr)
struct tm *Timeptr;
```

```
double difftime( Time1,  Time0)
time_t Time0, Time1;
```

```
char *asctime ( Tm)
const struct tm *Tm;
```

```
void tzset ( )
extern long int timezone;
extern int daylight;
extern char *tzname[];
```

Description

Attention: Do not use the **tzset** subroutine when linking with both **libc.a** and **libbsd.a**. The **tzset** subroutine sets the global external variable called **timezone**, which conflicts with the **timezone** subroutine in **libbsd.a**. This name collision may cause unpredictable results.

Attention: Do not use the **ctime**, **localtime**, **gmtime**, or **asctime** subroutine in a multithreaded environment. See the multithread alternatives in the **ctime_r**, **localtime_r**, **gmtime_r**, or **asctime_r** subroutine article.

The **ctime** subroutine converts a time value pointed to by the *Clock* parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into a 26-character string in the following form:

```
Sun Sept 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **ctime** subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The **localtime** subroutine converts the long integer pointed to by the *Clock* parameter, which contains the time in seconds since 00:00:00 UTC, 1 January 1970, into a **tm** structure. The **localtime** subroutine adjusts for the time zone and for daylight-saving time, if it is in effect. Use the time-zone information as though **localtime** called **tzset**.

The **gmtime** subroutine converts the long integer pointed to by the *Clock* parameter into a **tm** structure containing the Coordinated Universal Time (UTC), which is the time standard the operating system uses.

Note: UTC is the international time standard intended to replace GMT.

The **tm** structure is defined in the **time.h** file, and it contains the following members:

```
int tm_sec;     /* Seconds (0 - 59) */
int tm_min;     /* Minutes (0 - 59) */
int tm_hour;    /* Hours (0 - 23) */
int tm_mday;    /* Day of month (1 - 31) */
int tm_mon;     /* Month of year (0 - 11) */
int tm_year;    /* Year - 1900 */
int tm_wday;    /* Day of week (Sunday = 0) */
int tm_yday;    /* Day of year (0 - 365) */
int tm_isdst;   /* Nonzero = Daylight saving time */
```

The **mktime** subroutine is the reverse function of the **localtime** subroutine. The **mktime** subroutine converts the **tm** structure into the time in seconds since 00:00:00 UTC, 1 January 1970. The `tm_wday` and `tm_yday` fields are ignored, and the other components of the **tm** structure are not restricted to the ranges specified in the **/usr/include/time.h** file. The value of the `tm_isdst` field determines the following actions of the **mktime** subroutine:

Item	Description
0	Initially presumes that Daylight Saving Time (DST) is not in effect.
>0	Initially presumes that DST is in effect.
-1	Actively determines whether DST is in effect from the specified time and the local time zone. Local time zone information is set by the **tzset** subroutine.

Upon successful completion, the **mktime** subroutine sets the values of the `tm_wday` and `tm_yday` fields appropriately. Other fields are set to represent the specified time since January 1, 1970. However, the values are forced to the ranges specified in the **/usr/include/time.h** file. The final value of the `tm_mday` field is not set until the values of the `tm_mon` and `tm_year` fields are determined.

Note: The **mktime** subroutine cannot convert time values before 00:00:00 UTC, January 1, 1970 and after 03:14:07 UTC, January 19, 2038.

The **difftime** subroutine computes the difference between two calendar times: the *Time1* and -*Time0* parameters.

The **asctime** subroutine converts a **tm** structure to a 26-character string of the same format as **ctime**.

If the **TZ** environment variable is defined, then its value overrides the default time zone, which is the U.S. Eastern time zone. The **environment** facility contains the format of the time zone information specified by **TZ**. **TZ** is usually set when the system is started with the value that is defined in either the **/etc/environment** or **/etc/profile** files. However, it can also be set by the user as a regular environment variable for performing alternate time zone conversions.

The **tzset** subroutine sets the **timezone**, **daylight**, and **tzname** external variables to reflect the setting of **TZ**. The **tzset** subroutine is called by **ctime** and **localtime**, and it can also be called explicitly by an application program.

The **timezone** external variable contains the difference, in seconds, between UTC and local standard time. For example, the value of **timezone** is 5 * 60 * 60 for U.S. Eastern Standard Time.

The **daylight** external variable is nonzero when a daylight-saving time conversion should be applied. By default, this conversion follows the standard U.S. conventions; other conventions can be specified. The default conversion algorithm adjusts for the peculiarities of U.S. daylight saving time in 1974 and 1975.

The **tzname** external variable contains the name of the standard time zone (**tzname[0]**) and of the time zone when Daylight Saving Time is in effect (**tzname[1]**). For example:

```
char *tzname[2] = {"EST", "EDT"};
```

The **time.h** file contains declarations of all these subroutines and externals and the **tm** structure.

Parameters

Item	Description
Clock	Specifies the pointer to the time value in seconds.
Timeptr	Specifies the pointer to a **tm** structure.
Time1	Specifies the pointer to a **time_t** structure.
Time0	Specifies the pointer to a **time_t** structure.
Tm	Specifies the pointer to a **tm** structure.

Return Values

Attention: The return values point to static data that is overwritten by each call.

The **tzset** subroutine returns no value.

The **mktime** subroutine returns the specified time in seconds encoded as a value of type **time_t**. If the time cannot be represented, the function returns the value (**time_t**)-1.

The **localtime** and **gmtime** subroutines return a pointer to the **struct tm**.

The **ctime** and **asctime** subroutines return a pointer to a 26-character string.

The **difftime** subroutine returns the difference expressed in seconds as a value of type **double**.

ctime64, localtime64, gmtime64, mktime64, difftime64, or asctime64 Subroutine

Purpose

Converts the formats of date and time representations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>


char *ctime64 (Clock)
const time64_t *Clock;

struct tm *localtime64 (Clock)
const time64_t *Clock;

struct tm *gmtime64 (Clock)
const time64_t *Clock;
```

```
time64_t mktime64(Timeptr)
struct tm *Timeptr;

double difftime64(Time1, Time0)
time64_t Time0, Time1;

char *asctime64 (Tm)
const struct tm *Tm;
```

Description

Attention: Do not use the **ctime**, **localtime**, **gmtime**, or **asctime** subroutine in a multithreaded environment.

The **ctime64** subroutine converts a time value pointed to by the *Clock* parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into a 26-character string in the following form:

```
Sun Sept 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **ctime64** subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The **localtime64** subroutine converts the 64 bit long pointed to by the *Clock* parameter, which contains the time in seconds since 00:00:00 UTC, 1 January 1970, into a *tm* structure. The **localtime64** subroutine adjusts for the time zone and for daylight saving time, if it is in effect. Use the time-zone information as though **localtime64** called **tzset**.

The **gmtime64** subroutine converts the 64 bit long pointed to by the *Clock* parameter into a *tm* structure containing the Coordinated Universal Time (UTC), which is the time standard that the operating system uses.

Note: UTC is the international time standard intended to replace GMT.

The **mktime64** subroutine is the reverse function of the **localtime64** subroutine. The **mktime64** subroutine converts the *tm* structure into the time in seconds since 00:00:00 UTC, 1 January 1970. The **tm_wday** and **tm_yday** fields are ignored, and the other components of the *tm* structure are not restricted to the ranges specified in the **/usr/include/time.h** file. The value of the **tm_isdst** field determines the following actions of the **mktime64** subroutine:

Item	Description
0	Initially presumes that Daylight Saving Time (DST) is not in effect.
>0	Initially presumes that DST is in effect.
-1	Actively determines whether DST is in effect from the specified time and the local time zone. Local time zone information is set by the **tzset** subroutine.

Upon successful completion, the **mktime64** subroutine sets the values of the **tm_wday** and **tm_yday** fields appropriately. Other fields are set to represent the specified time since January 1, 1970. However, the values are forced to the ranges specified in the **/usr/include/time.h** file. The final value of the **tm_mday** field is not set until the values of the **tm_mon** and **tm_year** fields are determined.

Note: The **mktime64** subroutine cannot convert time values before 00:00:00 UTC, January 1, 1970 and after 23:59:59 UTC, December 31, 9999.

Note: The **difftime64** subroutine computes the difference between two calendar times: the *Time1* and *Time0* parameters.

Note: The **asctime64** subroutine converts a *tm* structure to a 26-character string of the same format as **ctime64**.

Parameters

Item	Description
Clock	Specifies the pointer to the time value in seconds.
Timeptr	Specifies the pointer to a **tm** structure.
Time1	Specifies the pointer to a **time64_t** structure.
Time0	Specifies the pointer to a **time64_t** structure.
Tm	Specifies the pointer to a **tm** structure.

Return Values

Attention: The return values point to static data that is overwritten by each call.

The **mktime64** subroutine returns the specified time in seconds encoded as a value of type **time64_t**. If the time cannot be represented, the function returns the value **(time64_t)-1**.

The **localtime64** and **gmtime64** subroutines return a pointer to the **tm** struct .

The **ctime64** and **asctime64** subroutines return a pointer to a 26-character string.

The **difftime64** subroutine returns the difference expressed in seconds as a value of type long double.

ctime64_r, localtime64_r, gmtime64_r, or asctime64_r Subroutine

Purpose

Converts the formats of date and time representations.

Library

Thread-Safe C Library (**libc_r.a**)

Syntax

```
#include <time.h>


char *ctime64_r(Timer, BufferPointer)
const time64_t * Timer;
char * BufferPointer;

struct tm *localtime64_r(Timer, CurrentTime)
const time64_t * Timer;
struct tm * CurrentTime;

struct tm *gmtime64_r (Timer, XTime)
const time64_t * Timer;
struct tm * XTime;

char *asctime64_r (TimePointer, BufferPointer)
const struct tm * TimePointer;
char * BufferPointer;
```

Description

The **ctime64_r** subroutine converts a time value pointed to by the *Timer* parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into the character array pointed to by the *BufferPointer* parameter. The character array should have a length of at least 26

characters so the converted time value fits without truncation. The converted time value string takes the form of the following example:

```
Sun Sept 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here. Thus, *ctime* will only return dates up to December 31, 9999.

The **ctime64_r** subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The **localtime64_r** subroutine converts the **time64_t** structure pointed to by the *Timer* parameter, which contains the time in seconds since 00:00:00 UTC, January 1, 1970, into the **tm** structure pointed to by the *CurrentTime* parameter. The **localtime64_r** subroutine adjusts for the time zone and for daylight saving time, if it is in effect.

The **gmtime64_r** subroutine converts the **time64_t** structure pointed to by the *Timer* parameter into the **tm** structure pointed to by the *XTime* parameter.

The **tm** structure is defined in the **time.h** header file. The **time.h** file contains declarations of these subroutines, externals, and the **tm** structure.

The **asctime64_r** subroutine converts the **tm** structure pointed to by the *TimePointer* parameter into a 26-character string in the same format as the **ctime64_r** subroutine. The results are placed into the character array, *BufferPointer*. The *BufferPointer* parameter points to the resulting character array, which takes the form of the following example:

```
Sun Sept 16 01:03:52 1973\n\0
```

Programs using this subroutine must link to the **libpthreads.a** library.

Parameters

Item	Description
Timer	Points to a **time64_t** structure, which contains the number of seconds since 00:00:00 UTC, January 1, 1970.
BufferPointer	Points to a character array at least 26 characters long.
CurrentTime	Points to a **tm** structure. The result of the **localtime64_r** subroutine is placed here.
XTime	Points to a **tm** structure used for the results of the **gmtime64_r** subroutine.
TimePointer	Points to a **tm** structure used as input to the **asctime64_r** subroutine.

Return Values

The **localtime64_r** and **gmtime64_r** subroutines return a pointer to the **tm** structure. The **asctime64_r** returns NULL if either *TimePointer* or *BufferPointer* is NULL.

The **ctime64_r** and **asctime64_r** subroutines return a pointer to a 26-character string. The **ctime64_r** subroutine returns NULL if the *BufferPointer* is NULL.

The **difftime64** subroutine returns the difference expressed in seconds as a value of type long double.

Files

Item	Description
/usr/include/time.h	Defines time macros, data types, and structures.

ctime_r, localtime_r, gmtime_r, or asctime_r Subroutine

Purpose

Converts the formats of date and time representations.

Library

Thread-Safe C Library (**libc_r.a**)

Syntax

```
#include <time.h>

char *ctime_r(Timer, BufferPointer)
const time_t * Timer;
char * BufferPointer;

struct tm *localtime_r(Timer, CurrentTime)
const time_t * Timer;
struct tm * CurrentTime;

struct tm *gmtime_r(Timer, XTime)
const time_t * Timer;
struct tm * XTime;

char *asctime_r(TimePointer, BufferPointer)
const struct tm * TimePointer;
char * BufferPointer;
```

Description

The **ctime_r** subroutine converts a time value pointed to by the *Timer* parameter, which represents the time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, into the character array pointed to by the *BufferPointer* parameter. The character array should have a length of at least 26 characters so the converted time value fits without truncation. The converted time value string takes the form of the following example:

```
Sun Sep 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **ctime_r** subroutine adjusts for the time zone and daylight saving time, if it is in effect.

The **localtime_r** subroutine converts the **time_t** structure pointed to by the *Timer* parameter, which contains the time in seconds since 00:00:00 UTC, January 1, 1970, into the **tm** structure pointed to by the *CurrentTime* parameter. The **localtime_r** subroutine adjusts for the time zone and for daylight saving time, if it is in effect.

The **gmtime_r** subroutine converts the **time_t** structure pointed to by the *Timer* parameter into the **tm** structure pointed to by the *XTime* parameter.

The **tm** structure is defined in the **time.h** header file. The **time.h** file contains declarations of these subroutines, externals, and the **tm** structure.

The **asctime_r** subroutine converts the **tm** structure pointed to by the *TimePointer* parameter into a 26-character string in the same format as the **ctime_r** subroutine. The results are placed into the character array, *BufferPointer*. The *BufferPointer* parameter points to the resulting character array, which takes the form of the following example:

```
Sun Sep 16 01:03:52 1973\n\0
```

Programs using this subroutine must link to the **libpthreads.a** library.

Parameters

Item	Description
Timer	Points to a **time_t** structure, which contains the number of seconds since 00:00:00 UTC, January 1, 1970.
BufferPointer	Points to a character array at least 26 characters long.
CurrentTime	Points to a **tm** structure. The result of the **localtime_r** subroutine is placed here.
XTime	Points to a **tm** structure used for the results of the **gmtime_r** subroutine.
TimePointer	Points to a **tm** structure used as input to the **asctime_r** subroutine.

Return Values

The **localtime_r** and **gmtime_r** subroutines return a pointer to the **tm** structure. The **asctime_r** returns NULL if either TimePointer or BufferPointer are NULL.

The **ctime_r** and **asctime_r** subroutines return a pointer to a 26-character string. The **ctime_r** subroutine returns NULL if the BufferPointer is NULL.

Files

Item	Description
/usr/include/time.h	Defines time macros, data types, and structures.

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, or isascii Subroutines

Purpose

Classifies characters.

Library

Standard Character Library (**libc.a**)

Syntax

```
#include <ctype.h>
```

int isalpha (*Character*)
int *Character*;

```
int isupper (Character)
int Character;
```

```
int islower (Character)
int Character;
```

```
int isdigit (Character)
int Character;
```

```
int isxdigit (Character)
int Character;
```

```
int isalnum (Character)
int Character;
```

```
int isspace (Character)
int Character;
```

```
int ispunct (Character)
int Character;
```

```
int isprint (Character)
int Character;
```

```
int isgraph (Character)
int Character;
```

```
int iscntrl (Character)
int Character;
```

```
int isascii (Character)
int Character;
```

Description

The **ctype** subroutines classify character-coded integer values specified in a table. Each of these subroutines returns a nonzero value for True and 0 for False.

Note: The **ctype** subroutines should only be used on character data that can be represented by a single byte value (0 through 255). Attempting to use the **ctype** subroutines on multi-byte locale data may give inconsistent results. Wide character classification routines (such as **iswprint**, **iswlower**, etc.) should be used with dealing with multi-byte character data.

Locale Dependent Character Tests

The following subroutines return nonzero (True) based upon the character class definitions for the current locale.

Item	Description
isalnum	Returns nonzero for any character for which the **isalpha** or **isdigit** subroutine would return nonzero. The **isalnum** subroutine tests whether the character is of the **alpha** or **digit** class.
isalpha	Returns nonzero for any character for which the **isupper** or **islower** subroutines would return nonzero. The **isalpha** subroutine also returns nonzero for any character defined as an alphabetic character in the current locale, or for a character for which *none* of the **iscntrl**, **isdigit**, **ispunct**, or **isspace** subroutines would return nonzero. The **isalpha** subroutine tests whether the character is of the **alpha** class.
isupper	Returns nonzero for any uppercase letter [A through Z]. The **isupper** subroutine also returns nonzero for any character defined to be uppercase in the current locale. The **isupper** subroutine tests whether the character is of the **upper** class.
islower	Returns nonzero for any lowercase letter [a through z]. The **islower** subroutine also returns nonzero for any character defined to be lowercase in the current locale. The **islower** subroutine tests whether the character is of the **lower** class.
isspace	Returns nonzero for any white-space character (space, form feed, new line, carriage return, horizontal tab or vertical tab). The **isspace** subroutine tests whether the character is of the **space** class.

Item	Description
ispunct	Returns nonzero for any character for which the **isprint** subroutine returns nonzero, except the space character and any character for which the **isalnum** subroutine would return nonzero. The **ispunct** subroutine also returns nonzero for any locale-defined character specified as a punctuation character. The **ispunct** subroutine tests whether the character is of the **punct** class.
isprint	Returns nonzero for any printing character. Returns nonzero for any locale-defined character that is designated a printing character. This routine tests whether the character is of the **print** class.
isgraph	Returns nonzero for any character for which the **isprint** character returns nonzero, except the space character. The **isgraph** subroutine tests whether the character is of the **graph** class.
iscntrl	Returns nonzero for any character for which the **isprint** subroutine returns a value of False (0) and any character that is designated a control character in the current locale. For the C locale, control characters are the ASCII delete character (0127 or 0x7F), or an ordinary control character (less than 040 or 0x20). The **iscntrl** subroutine tests whether the character is of the **cntrl** class.

Locale Independent Character Tests

The following subroutines return nonzero for the same characters, regardless of the locale:

Item	Description
isdigit	*Character* is a digit in the range [0 through 9].
isxdigit	*Character* is a hexadecimal digit in the range [0 through 9], [A through F], or [a through f].
isascii	*Character* is an ASCII character with a value in the range [0 through 0x7F].

Parameter

Item	Description
Character	Indicates the character to be tested (integer value).

Return Codes

The **ctype** subroutines return nonzero (True) if the character specified by the *Character* parameter is a member of the selected character class; otherwise, a 0 (False) is returned.

cuserid Subroutine

Purpose

Gets the alphanumeric user name associated with the current process.

Library

Standard C Library (**libc.a**)

Use the **libc_r.a** library to access the thread-safe version of this subroutine.

Syntax

```
#include <stdio.h>
```

```
char *cuserid ( Name)
char *Name;
```

Description

The **cuserid** subroutine gets the alphanumeric user name associated with the current process. This subroutine generates a character string representing the name of a process's owner.

Note: The **cuserid** subroutine duplicates functionality available with the **getpwuid** and **getuid** subroutines. Present applications should use the **getpwuid** and **getuid** subroutines.

If the *Name* parameter is a null pointer, then a character string of size `L_cuserid` is dynamically allocated with **malloc**, and the character string representing the name of the process owner is stored in this area. The **cuserid** subroutine then returns the address of this area. Multithreaded application programs should use this functionality to obtain thread specific data, and then continue to use this pointer in subsequent calls to the **curserid** subroutine. In any case, the application program must deallocate any dynamically allocated space with the **free** subroutine when the data is no longer needed.

If the *Name* parameter is not a null pointer, the character string is stored into the array pointed to by the *Name* parameter. This array must contain at least the number of characters specified by the constant `L_cuserid`. This constant is defined in the **stdio.h** file.

If the user name cannot be found, the **cuserid** subroutine returns a null pointer; if the *Name* parameter is not a null pointer, a null character ('\0') is stored in *Name* [0].

Parameter

Item	Description
Name	Points to a character string representing a user name.

curs_set Subroutine

Purpose

Sets the cursor visibility.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int curs_set(int  visibility);
```

Description

The **curs_set** subroutine sets the appearance of the cursor based on the value of visibility:

```
Value of visibility   Appearance of Cursor
```

Item	Description
0	invisible
1	terminal-specific normal mode

Item	Description
2	terminal-specific high visibility mode

The terminal does not necessarily support all the above values.

Parameters

Item	Description
Visibility	Sets the cursor state. You can set the cursor state to one of the following: **0** Invisible **1** Visible **2** Very visible

Return Values

If the terminal supports the cursor mode specified by *visibility*, then the **cur_set** subroutine returns the previous cursor state. Otherwise, the subroutine returns ERR.

Examples

To set the cursor state to invisible, use:

```
curs_set(0);
```

c16rtomb, c32rtomb Subroutine

Purpose

The **c16rtomb** and **c32rtomb** subroutines convert a 16-bit wide character (UTF-16) and a 32-bit wide character (UTF-32) to the corresponding multibyte character of the current locale.

Library

Standard C library (**libc.a**)

Syntax

```
#include <uchar.h>
size_t c16rtomb (char * restrict s,  char16_t c16,
 mbstate_t * restrict ps);

size_t c32rtomb (char * restrict s,  char32_t c32,
 mbstate_t * restrict ps);
```

Description

If the value of the **s** parameter is a null pointer, the **c16rtomb** subroutine is equivalent to the following call, where **buf** is an internal buffer.

```
c16rtomb(buf, L'\0', ps)
```

If the value of the **s** parameter is not a null pointer, the **c16rtomb** subroutine determines the number of bytes needed to represent the multibyte character that corresponds to the wide character specified by the **c16** parameter, including any shift sequences, and stores the multibyte character representation in an array, in which the first element is specified by the **s** parameter.

The value greater than the value of **MB_CUR_MAX** bytes is stored.

If the value of the **c16** parameter is a null wide character, a null byte is stored, preceded by any shift sequence that is needed to restore the initial shift state and the resulting state is described is the initial conversion state.

If the value of the **s** parameter is a null pointer, the **c32rtomb** subroutine is equivalent to the following call, where **buf** is an internal buffer.

```
c32rtomb(buf, L'\0', ps)
```

If the value of the **s** parameter is not a null pointer, the **c32rtomb** subroutine determines the number of bytes needed to represent the multibyte character that corresponds to the wide character specified by the **c32** parameter, including any shift sequences, and stores the multibyte character representation in an array, in which the first element is specified by the **s** parameter.

The value greater than the value of **MB_CUR_MAX** bytes is stored. If the value of the **c32** parameter is a null wide character, a null byte is stored, preceded by any shift sequence that is needed to restore the initial shift state and the resulting state is described is the initial conversion state.

Note: The **c16rtomb** and **c32rtomb** subroutines include the **ps** parameter which is of the type pointer to **mbstate_t** value that points to an object which describes the current conversion state of the associated multibyte character sequence, which the subroutines alter as necessary. If the **ps** parameter is a null pointer, each subroutine uses its own internal **mbstate_t** object. The **c16rtomb** and **c32rtomb** subroutines do not avoid data races with other calls to the same subroutine.

Parameters

Item	Description
s	Specifies the first element of an array where the multibyte character representation is stored.
c16, c32	Represents the wide character sequence.
ps	Specifies the state of the multibyte conversion.

Example

- The **mbstate_t** pointer can be used as follows:

```
 mbstate_t ss = 0;
```

```
int x = c16rtomb(out, in, &ss);
```

Return Values

The **c16rtomb** subroutine returns the number of bytes stored in an array object, including any shift sequences.

When the value of the **c16** parameter is not a valid wide character, an encoding error occurs. The function stores the value of the **EILSEQ** macro in the **errno** variable and returns the *(size_t)(-1)*. The conversion state is unspecified.

The **c32rtomb** subroutine returns the number of bytes stored in an array object, including any shift sequences.

When the value of the **c32** parameter is not a valid wide character, an encoding error occurs. The function stores the value of the **EILSEQ** macro in the **errno** variable and returns *(size_t)(-1)*. The conversion state is unspecified.

Error codes

The **c16rtomb** and **c32rtomb** subroutine is unsuccessful if the following error code is set.

Item	Description
EILSEQ	Indicates an invalid multibyte character sequence.

Files

The **uchar.h** file defines standard macros, data types, and subroutines.

d

The following Base Operating System (BOS) runtime services begin with the letter *d*.

def_prog_mode, def_shell_mode, reset_prog_mode or reset_shell_mode Subroutine

Purpose

Saves/restores the program or shell terminal modes.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int def_prog_mode
(void);

int def_shell_mode
(void);

int reset_prog_mode
(void);

int reset_shell_mode
(void);
```

Description

The **def_prog_mode** subroutine saves the current terminal modes as the "program" (in Curses) state for use by the **reset_prog_mode** subroutine.

The **def_shell_mode** subroutine saves the current terminal modes as the "shell" (not in Curses) state for use by the **reset_shell_mode** subroutine.

The **reset_prog_mode** subroutine restores the terminal to the "program" (in Curses) state.

The **reset_shell_mode** subroutine restores the terminal to the "shell" (not in Curses) state.

These subroutines affect the mode of the terminal associated with the current screen.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

For the **def_prog_mode** subroutine:

To save the "**in curses**" state, enter:

```
def_prog_mode();
```

For the **def_shell_mode** subroutine:

To save the "**out of curses**" state, enter:

```
def_shell_mode();
```

This routine saves the "**out of curses**" state.

def_shell_mode Subroutine

Purpose

Saves the current terminal modes as shell mode ("out of curses").

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
def_shell_mode( )
```

Description

The **def_shell_mode** subroutine saves the current terminal driver line discipline modes in the current terminal structure for later use by **reset_shell_mode()**. The **def_shell_mode** subroutine is called automatically by the **setupterm** subroutine.

This routine would normally not be called except by a library routine.

Example

To save the "**out of curses**" state, enter:

```
def_shell_mode();
```

This routine saves the "**out of curses**" state.

defssys Subroutine

Purpose

Initializes the **SRCsubsys** structure with default values.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
```

void defssys(*SRCSubsystem***)**
struct SRCsubsys ****SRCSubsystem*;**

Description

The **defssys** subroutine initializes the **SRCsubsys** structure of the **/usr/include/sys/srcobj.h** file with the following default values:

Field	Value
display	SRCYES
multi	SRCNO
contact	SRCSOCKET
waittime	TIMELIMIT
priority	20
action	ONCE
standerr	**/dev/console**
standin	**/dev/console**
standout	**/dev/console**

All other numeric fields are set to 0, and all other alphabetic fields are set to an empty string.

This function must be called to initialize the **SRCsubsys** structure before an application program uses this structure to add records to the subsystem object class.

Parameters

Item	Description
SRCSubsystem	Points to the **SRCsubsys** structure.

del_curterm, restartterm, set_curterm, or setupterm Subroutine

Purpose

Interfaces to the **terminfo** database.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <term.h>

int del_curterm(TERMINAL *oterm);

int restartterm(char *term,
int fildes,
int *erret);

TERMINAL *set_curterm(TERMINAL *nterm);

int setupterm(char *term,
int fildes,
int *erret);
```

Description

The **del_curterm, restartterm**, **set_curterm**, **setupterm** subroutines retrieve information from the **terminfo** database.

To gain access to the **terminfo** database, the **setupterm** subroutine must be called first. It is automatically called by the **initscr** and **newterm** subroutines. The **setupterm** subroutine initialises the other subroutines to use the **terminfo** record for a specified terminal (which depends on whether the **use_env** subroutine was called). It sets the dur_term external variable to a TERMINAL structure that contains the record from the terminfo database for the specified terminal.

The terminal type is the character string term; if term is a null pointer, the environment variable TERM is used. If TERM is not set or if its value is an empty string, the "unknown" is used as the terminal type. The application must set the *fildes* parameter to a file descriptor, open for output, to the terminal device, before calling the **setupterm** subroutine. If the *erret* parameter is not null, the integer it points to is set to one of the following values to report the function outcome:

Item	Description
-1	The terminfo database was not found (function fails).
0	The entry for the terminal was not found in **terminfo** (function fails).
1	Success.

A simple call to the **setupterm** subroutine that uses all the defaults and sends the output to stdout is:

```
setupterm(char *)0, fileno(stdout), (int *)0);
```

The **set_curterm** subroutine sets the variable **cur_term** to *nterm*, and makes all of the terminfo boolean, numeric, and string variables use the values from *nterm*.

The **del_curterm** subroutine frees the space pointed to by oterm and makes it available for further use. If *oterm* is the same as **cur_term**, references to any of the terminfo boolean, numeric, and string variables thereafter may refer to invalid memory locations until the **setupterm** subroutine is called again.

The **restartterm** subroutine assumes a previous call to the **setupterm** subroutine (perhaps from the **initscr** or **newterm** subroutine). It lets the application specify a different terminal type in *term* and updates the information returned by the **baudrate** subroutine based on the *fildes* parameter, but does not destroy other information created by the **initscr**, **newterm**, or **setupterm** subroutines.

Parameters

Item	Description
**oterm*	
**term*	
fildes	
**erret*	
**nterm*	

Return Values

Upon successful completion, the **set_curterm** subroutine returns the previous value of cur_term. Otherwise, it returns a null pointer.

Upon successful completion, the other subroutines return OK. Otherwise, they return ERR.

Examples

To free the space occupied by a **TERMINAL** structure called `my_term`, use:

```
TERMINAL *my_term; del_curterm(my_term);
```

For the **restartterm** subroutine:

To restart an **aixterm** after a previous memory save and exit on error with a message, enter:

```
restartterm("aixterm", 1, (int*)0);
```

For the **set_curterm** subroutine:

To set the **cur_term** variable to point to the `my_term` terminal, use:

```
TERMINAL *newterm; set_curterm(newterm);
```

For the **setupterm** subroutine:

To determine the current terminal's capabilities using **$TERM** as the terminal name, standard output as output, and returning no error codes, enter:

```
setupterm((char*) 0, 1, (int*) 0);
```

delay_output Subroutine

Purpose

Sets the delay output.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

int delay_output(int *ms*);

Description

On terminals that support pad characters, the **delay_output** subroutine pauses the output for at least *ms* milliseconds. Otherwise, the length of the delay is unspecified.

Parameters

Item	Description
ms	Specifies the number of milliseconds to delay output.

Return Values

Upon successful completion, the **delay_output** subroutine returns OK. Otherwise, it returns ERR.

Examples

To set the output to delay 250 milliseconds, enter:

```
delay_output(250);
```

delch, mvdelch, mvwdelch or wdelch Subroutine

Purpose

Deletes the character from a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int delch(void);
```

```
int mvdelch
(int y
int x);
```

```
mvwdelch
(WINDOW  *win;
int y
int x);
```

```
wdelch
(WINDOW *win);
```

Description

The **delch**, **mvdelch**, **mvwdelch**, and **wdelch** subroutines delete the character at the current or specified position in the current or specified window. This subroutine does not change the cursor position.

Parameters

Item	Description
x	
y	
**win*	Identifies the window from which to delete the character.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To delete the character at the current cursor location in the standard screen structure, enter:

   ```
   mvdelch();
   ```

2. To delete the character at cursor position y=20 and x=30 in the standard screen structure, enter:

```
mvwdelch(20, 30);
```

3. To delete the character at cursor position y=20 and x=30 in the user-defined window my_window, enter:

```
wdelch(my_window, 20, 30);
```

deleteln or wdeleteln Subroutine

Purpose

Deletes lines in a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int deleteln(void);
```

int wdeleteln(WINDOW **win*);

Description

The **deleteln** and **wdeleteln** subroutines delete the line containing the cursor in the current or specified window and move all lines following the current line one line toward the cursor. The last line of the window is cleared. The cursor position does not change.

Parameters

Item	Description
**win*	Specifies the window in which to delete the line.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To delete the current line in stdscr, enter:

```
deleteln();
```

2. To delete the current line in the user-defined window my_window, enter:

```
WINDOW *my_window;
wdeleteln(my_window);
```

delwin Subroutine

Purpose

Deletes a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int delwin(WINDOW  *win);
```

Description

The **delwin** subroutine deletes *win*, freeing all memory associated with it. The application must delete subwindows before deleting the main window.

Parameters

Item	Description
**win*	Specifies the window to delete.

Return Values

Upon successful completion, the **delwin** subroutine returns OK. Otherwise, it returns ERR.

Examples

To delete the user-defined window `my_window` and its subwindow `my_sub_window`, enter:

```
WINDOW *my_sub_window, *my_window;
delwin(my_sub_window);

delwin(my_window);
```

delssys Subroutine

Purpose

Removes the subsystem objects associated with the *SubsystemName* parameter.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
```

```
int delssys ( SubsystemName)
char *SubsystemName;
```

Description

The **delssys** subroutine removes the subsystem objects associated with the specified subsystem. This removes all objects associated with that subsystem from the following object classes:

- Subsystem
- Subserver Type
- Notify

The program running with this subroutine must be running with the group **system**.

Parameter

Item	Description
SubsystemName	Specifies the name of the subsystem.

Return Values

Upon successful completion, the **delssys** subroutine returns a positive value. If no record is found, a value of 0 is returned. Otherwise, -1 is returned and the **odmerrno** variable is set to indicate the error. See "Appendix B. ODM Error Codes" for a description of possible **odmerrno** values.

Security

Privilege Control:

SET_PROC_AUDIT kernel privilege

Files Accessed:

Mode	File
644	**/etc/objrepos/SRCsubsys**
644	**/etc/objrepos/SRCsubsvr**
644	**/etc/objrepos/SRCnotify**

Auditing Events:

Event	Information
SRC_Delssys	Lists in an audit log the name of the subsystem being removed.

Files

Item	Description
/etc/objrepos/SRCsubsys	SRC Subsystem Configuration object class.
/etc/objrepos/SRCsubsvr	SRC Subsystem Configuration object class.
/etc/objrepos/SRCnotify	SRC Notify Method object class.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.
/usr/include/sys/srcobj.h	Defines object structures used by the SRC.
/usr/include/spc.h	Defines external interfaces provided by the SRC subroutines.

derwin, newwin, or subwin Subroutine

Purpose

Window creation subroutines.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

WINDOW *derwin(WINDOW *orig,
int nlines,
int ncols,
int begin_y,
int begin_x);

WINDOW *newwin(int nlines,
int ncols,
int begin_y,
 int begin_x);

WINDOW *subwin(WINDOW *orig,
int nlines,
int ncols,
int begin_y,
int begin_x);
```

Description

The **derwin** subroutine is the same as the **subwin** subroutine except that *begin_y* and *begin_x* are relative to the origin of the window *orig* rather than absolute screen positions.

The **newwin** subroutine creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is at (*begin_y*, *begin_x*). If *nlines* is zero, it defaults to LINES - *begin_y*; if *ncols* is zero, it defaults to COLS - *begin_x*.

The **subwin** subroutine creates a new window with *nlines* lines and *ncols* columns, positioned so that the origin is at (*begin_y*, *begin_x*). (This position is an absolute screen position, not a position relative to the window orig.) If any part of the new window is outside *orig*, the subroutine fails and the window is not created.

Parameters

Item	Description
ncols	
nlines	
begin_y	
begin_x	

Return Values

Upon successful completion, these subroutines return a pointer to the new window. Otherwise, they return a null pointer.

Examples

For the **derwin** and **newwin** subroutines:

1. To create a new window, enter:

```
WINDOW *my_window;

my_window = newwin(5, 10, 20, 30);
```

   my_window is now a window 5 lines deep, 10 columns wide, starting at the coordinates y = 20, x = 30. That is, the upper left corner is at coordinates y = 20, x = 30, and the lower right corner is at coordinates y = 24, x = 39.

2. To create a window that is flush with the right side of the terminal, enter:

```
WINDOW *my_window;

my_window = newwin(5, 0, 20, 30);
```

   my_window is now a window 5 lines deep, extending all the way to the right side of the terminal, starting at the coordinates y = 20, x = 30. The upper left corner is at coordinates y = 20, x = 30, and the lower right corner is at coordinates y = 24, x = lastcolumn.

3. To create a window that fills the entire terminal, enter:

```
WINDOW *my_window;

my_window = newwin(0, 0, 0, 0);
```

   my_window is now a screen that is a window that fills the entire terminal's display.

For the **subwin** subroutine:

1. To create a subwindow, use:

```
WINDOW *my_window, *my_sub_window;
my_window = newwin (“derwin, newwin, or subwin Subroutine” on page 246)
            (5, 10, 20, 30);
```

   my_sub_window is now a subwindow 2 lines deep, 5 columns wide, starting at the same coordinates of its parent window my_window. That is, the subwindow's upper-left corner is at coordinates y = 20, x = 30 and lower-right corner is at coordinates y = 21, x = 34.

2. To create a subwindow that is flush with the right side of its parent, use

```
WINDOW *my_window, *my_sub_window;
my_window =
newwin (“derwin, newwin, or subwin Subroutine” on page 246)(5, 10, 20, 30);
my_sub_window = subwin(my_window, 2, 0, 20, 30);
```

   my_sub_window is now a subwindow 2 lines deep, extending all the way to the right side of its parent window my_window, and starting at the same coordinates. That is, the subwindow's upper-left corner is at coordinates y = 20, x = 30 and lower-right corner is at coordinates y = 21, x = 39.

3. To create a subwindow in the lower-right corner of its parent, use:

```
WINDOW *my_window, *my_sub_window
my_window = newwwin (“derwin, newwin, or subwin Subroutine” on page 246)
            (5, 10, 20, 30);
my_sub_window = subwin(my_window, 0, 0, 22, 35);
```

   my_sub_window is now a subwindow that fills the bottom right corner of its parent window, my_window, starting at the coordinates y = 22, x = 35. That is, the subwindow's upper-left corner is at coordinates y = 22, x = 35 and lower-right corner is at coordinates y = 24, x = 39.

dirname Subroutine

Purpose

Report the parent directory name of a file path name.

Library

Standard C Library **(libc.a)**

Syntax

#include <libgen.h>

char *dirname (*path*) **char ***path*

Description

Given a pointer to a character string that contains a file system path name, the **dirname** subroutine returns a pointer to a string that is the parent directory of that file. Trailing "/" characters in the path are not counted as part of the path.

If *path* is a null pointer or points to an empty string, a pointer to a static constant "." is returned.

The **dirname** and **basename** subroutines together yield a complete path name. **dirname** (*path*) is the directory where **basename** (*path*) is found.

Parameters

Item	Description
path	Character string containing a file system path name.

Return Values

The **dirname** subroutine returns a pointer to a string that is the parent directory of *path.* If *path* or **path* is a null pointer or points to an empty string, a pointer to a string "." is returned. The **dirname** subroutine may modify the string pointed to by *path* and may return a pointer to static storage that may then be overwritten by sequent calls to the **dirname** subroutine.

Examples

A simple file name and the strings "." and ".." all have "." as their return value.

Input string	Output string
/usr/lib	/usr
/usr/	/
usr	.
/	/
.	.
..	.

The following code reads a path name, changes directory to the appropriate directory, and opens the file.

```
char path [MAXPATHEN],   *pathcopy;
int fd;
fgets (path, MAXPATHEN, stdin);
```

```
pathcopy = strdup (path);
chdir (dirname (pathcopy) );
fd = open (basename (path), O_RDONLY);
```

disclaim and disclaim64 Subroutines

Purpose

Disclaim the content of a memory address range.

Syntax

```
#include <sys/shm.h>
int disclaim ( Address,  Length, Flag)
char *Address;
unsigned int Length, Flag;

int disclaim64( Address, Length, Flag)
void *Address;
size_t Length;
unsigned long Flag;
```

Description

The **disclaim** and **disclaim64** subroutines mark an area of memory having content that is no longer needed. The system then stops paging the memory area. These subroutines cannot be used on memory that is mapped to a file by the **shmat** subroutine.

Parameters

Item	Description
Address	Points to the beginning of the memory area.
Length	Specifies the length of the memory area in bytes.
Flag	Must be the **DISCLAIM_ZEROMEM** value, which indicates that each memory location in the address range should be set to zero.

Return Values

When successful, the **disclaim** and **disclaim64** subroutines return a value of 0.

Error Codes

If the **disclaim** and **disclaim64** subroutines are not successful, they returns a value of -1 and set the **errno** global variable to indicate the error. The **disclaim** and **disclaim64** subroutines are not successful if one or more of the following are true:

Item	Description
EFAULT	The calling process does not have write access to the area of memory that begins at the *Address* parameter and extends for the number of bytes specified by the *Length* parameter.
EINVAL	The value of the *Flag* parameter is not valid.
EINVAL	The memory area is mapped to a file.

dlclose Subroutine

Purpose

Closes and unloads a module loaded by the **dlopen** subroutine.

Syntax

```
#include <dlfcn.h>
```

```
int dlclose(Data);
void *Data;
```

Description

The **dlclose** subroutine is used to remove access to a module loaded with the **dlopen** subroutine. In addition, access to dependent modules of the module being unloaded is removed as well.

The `dlclose` subroutine performs C++ termination, like the **terminateAndUnload** subroutine does.

Modules being unloaded with the **dlclose** subroutine will not be removed from the process's address space if they are still required by other modules. Nevertheless, subsequent uses of *Data* are invalid, and further uses of symbols that were exported by the module being unloaded result in undefined behavior.

Parameters

Item	Description
Data	A loaded module reference returned from a previous call to **dlopen**.

Return Values

Upon successful completion, `0` (zero) is returned. Otherwise, **errno** is set to **EINVAL**, and the return value is also **EINVAL**. Even if the **dlclose** subroutine succeeds, the specified module may still be part of the process's address space if the module is still needed by other modules.

Error Codes

Item	Description
EINVAL	The *Data* parameter does not refer to a module opened by **dlopen** that is still open. The parameter may be corrupt or the module may have been unloaded by a previous call to **dlclose**.

dlerror Subroutine

Purpose

Returns a pointer to information about the last **dlopen**, **dlsym**, or **dlclose** error.

Syntax

```
#include <dlfcn.h>
```

```
char *dlerror(void);
```

Description

The **dlerror** subroutine is used to obtain information about the last error that occurred in a dynamic loading routine (that is, **dlopen** , **dlsym** , or **dlclose**). The returned value is a pointer to a null-terminated string without a final newline. Once a call is made to this function, subsequent calls without any intervening dynamic loading errors will return NULL.

Applications can avoid calling the **dlerror** subroutine, in many cases, by examining **errno** after a failed call to a dynamic loading routine. If **errno** is **ENOEXEC**, the **dlerror** subroutine will return additional information. In all other cases, **dlerror** will return the string corresponding to the value of **errno**.

The **dlerror** function may invoke **loadquery** to ascertain reasons for a failure. If a call is made to l**oad** or **unload** between calls to **dlopen** and **dlerror**, incorrect information may be returned.

Return Values

A pointer to a static buffer is returned; a NULL value is returned if there has been no error since the last call to **dlerror**. Applications should not write to this buffer; they should make a copy of the buffer if they wish to preserve the buffer's contents.

dlopen Subroutine

Purpose

Dynamically loads a module into the calling process.

Syntax

```
#include <dlfcn.h>
```

```
void *dlopen (FilePath, Flags);
const char *FilePath;
int Flags;
```

Description

The **dlopen** subroutine loads the module specified by *FilePath* into the executing process's address space. Dependents of the module are automatically loaded as well. If the module is already loaded, it is not loaded again, but a new, unique value will be returned by the **dlopen** subroutine.

The **dlopen** subroutine is a portable way of dynamically loading shared libraries. It performs C++ static initialization of the modules that it loads, like the **loadAndInit** subroutine does.

The value returned by the **dlopen** might be used in subsequent calls to **dlsym** and **dlclose**. If an error occurs during the operation, **dlopen** returns NULL.

If the main application was linked with the **-brtl** option, then the runtime linker is invoked by **dlopen**. If the module being loaded was linked with runtime linking enabled, both intra-module and inter-module references are overridden by any symbols available in the main application. If runtime linking was enabled, but the module was not built enabled, then all inter-module references will be overridden, but some intra-module references will not be overridden.

If the module being opened with **dlopen** or any of its dependents is being loaded for the first time, initialization routines for these newly-loaded routines are called (after runtime linking, if applicable) before **dlopen** returns. Initialization routines are the functions specified with the **-binitfini:** linker option when the module was built. (See the **ld** command for more information about this option.)

After calling the initialization functions for all newly-loaded modules, C++ static initialization is performed. If you call the **dlopen** subroutine from within an initialization function or a C++ static initialization function, modules loaded by the nested **dlopen** subroutine might be initialized before completely initializing the originally loaded modules.

If a **dlopen** subroutine is called from within a **binitfini** function, the initialization of the current module is abandoned for other modules.

Note: If the module being loaded has read-other permission, the module is loaded into the global shared library segment. Modules loaded into the global shared library segment are not unloaded even if they are no longer being used. Use the **slibclean** command to remove unused modules from the global shared library segment. To load the module in the process private region, unload the module completely using the **slibclean** command, and then unset its read-other permission.

The LIBPATH or LD_LIBRARY_PATH environment variables can be used to specify a list of directories in which the **dlopen** subroutine searches for the named module. The running application also contains a set of library search paths that were specified when the application was linked. The **dlopen** subroutine searches the modules based on the mechanism that the **load** subroutine defines, because the **dlopen** subroutine internally calls the **load** subroutine with the **L_LIBPATH_EXEC** flag.

Item	Description
FilePath	Specifies the name of a file containing the loadable module. This parameter can be contain an absolute path, a relative path, or no path component. If *FilePath* contains a slash character, *FilePath* is used directly, and no directories are searched. If the *FilePath* parameter is /unix, **dlopen** returns a value that can be used to look up symbols in the current kernel image, including those symbols found in any kernel extension that was available at the time the process began execution. If the value of *FilePath* is NULL, a value for the main application is returned. This allows dynamically loaded objects to look up symbols in the main executable, or for an application to examine symbols available within itself.

Flags

Specifies variations of the behavior of **dlopen**. Either **RTLD_NOW** or **RTLD_LAZY** must always be specified. Other flags may be OR'ed with **RTLD_NOW** or **RTLD_LAZY**.

Item	Description
RTLD_NOW	Load all dependents of the module being loaded and resolve all symbols.
RTLD_LAZY	Specifies the same behavior as **RTLD_NOW**. In a future release of the operating system, the behavior of the **RTLD_LAZY** may change so that loading of dependent modules is deferred of resolution of some symbols is deferred.
RTLD_GLOBAL	Allows symbols in the module being loaded to be visible when resolving symbols used by other **dlopen** calls. These symbols will also be visible when the main application is opened with **dlopen**(NULL, *mode*).
RTLD_LOCAL	Prevent symbols in the module being loaded from being used when resolving symbols used by other **dlopen** calls. Symbols in the module being loaded can only be accessed by calling **dlsym** subroutine. If neither **RTLD_GLOBAL** nor **RTLD_LOCAL** is specified, the default is **RTLD_LOCAL**. If both flags are specified, **RTLD_LOCAL** is ignored.
RTLD_MEMBER	The **dlopen** subroutine can be used to load a module that is a member of an archive. The **L_LOADMEMBER** flag is used when the **load** subroutine is called. The module name *FilePath* names the archive and archive member according to the rules outlined in the **load** subroutine.

Item	Description
RTLD_NOAUTODEFER	Prevents deferred imports in the module being loaded from being automatically resolved by subsequent loads. The **L_NOAUTODEFER** flag is used when the **load** subroutine is called. Ordinarily, modules built for use by the **dlopen** and **dlsym** sub routines will not contain deferred imports. However, deferred imports can be still used. A module opened with **dlopen** may provide definitions for deferred imports in the main application, for modules loaded with the **load** subroutine (if the **L_NOAUTODEFER** flag was not used), and for other modules loaded with the **dlopen** subroutine (if the **RTLD_NOAUTODEFER** flag was not used).

Return Values

Upon successful completion, **dlopen** returns a value that can be used in calls to the **dlsym** and **dlclose** subroutines. The value is not valid for use with the **loadbind** and **unload** subroutines.

If the **dlopen** call fails, NULL (a value of 0) is returned and the global variable **errno** is set. If **errno** contains the value ENOEXEC, further information is available via the **dlerror** function.

Error Codes

See the **load** subroutine for a list of possible **errno** values and their meanings.

dlsym Subroutine

Purpose

Looks up the location of a symbol in a module that is loaded with **dlopen**.

Syntax

```
#include <dlfcn.h>
```

```
void *dlsym(Handle, Symbol);
void *Handle;
const char *Symbol;
```

Description

The **dlsym** subroutine looks up a named symbol exported from a module loaded by a previous call to the **dlopen** subroutine. Only exported symbols are found by **dlsym**. See the **ld** command to see how to export symbols from a module.

Item	Description
Handle	Specifies a value returned by a previous call to **dlopen** or one of the special handles **RTLD_DEFAULT**, **RTLD_NEXT** or **RTLD_MYSELF**.
Symbol	Specifies the name of a symbol exported from the referenced module in the form of a NULL-terminated string or the special symbol name **RTLD_ENTRY**.

Note: C++ symbol names should be passed to **dlsym** in mangled form; **dlsym** does not perform any name demangling on behalf of the calling application.

In case of the special handle **RTLD_DEFAULT**, **dlsym** searches for the named symbol starting with the first module loaded. It then proceeds through the list of initial loaded modules and any global modules

obtained with **dlopen** until a match is found. This search follows the default model employed to relocate all modules within the process.

In case of the special handle **RTLD_NEXT**, **dlsym** searches for the named symbol in the modules that were loaded following the module from which the **dlsym** call is being made.

In case of the special handle **RTLD_MYSELF**, **dlsym** searches for the named symbol in the modules that were loaded starting with the module from which the **dlsym** call is being made.

In case of the special symbol name **RTLD_ENTRY**, **dlsym** returns the module's entry point. The entry point, if present, is the value of the module's loader section symbol marked as entry point.

In case of **RTLD_DEFAULT**, **RTLD_NEXT**, and **RTLD_MYSELF**, if the modules being searched have been loaded from **dlopen** calls, **dlsym** searches the module only if the caller is part of the same **dlopen** dependency hierarchy, or if the module was given global search access. See **dlopen** for a discussion of the **RTLD_GLOBAL** mode.

A search for the named symbol is based upon breadth-first ordering of the module and its dependants. If the module was constructed using the **-G** or **-brtl** linker option, the module's dependants will include all modules named on the **ld** command line, in the original order. The dependants of a module that was not linked with the **-G** or **-brtl** linker option will be listed in an unspecified order.

Return Values

If the named symbol is found, its address is returned. If the named symbol is not found, NULL is returned and **errno** is set to 0. If *Handle* or *Symbol* is invalid, NULL is returned and **errno** is set to **EINVAL** .

If the first definition found is an export of an imported symbol, this definition will satisfy the search. The address of the imported symbol is returned. If the first definition is a deferred import, the definition is ignored and the search continues.

If the named symbol refers to a BSS symbol (uninitialized data structure), the search continues until an initialized instance of the symbol is found or the module and all of its dependants have been searched. If an initialized instance is found, its address is returned; otherwise, the address of the first uninitialized instance is returned.

Error Codes

Item	Description
EINVAL	If the *Handle* parameter does not refer to a module opened by **dlopen** that is still loaded or if the *Symbol* parameter points to an invalid address, the **dlsym** subroutine returns NULL and **errno** is set to **EINVAL**.

dirfd Subroutine

Purpose

Extracts the file descriptor used by a DIR stream.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <dirent.h>
```

```
int dirfd(dirp);
DIR *dirp;
```

Description

The **dirfd** subroutine returns a file descriptor that refers to the directory pointed to by the *dirp* argument. This file descriptor is closed by a call to the **closedir** subroutine. If an attempt is made to close the file descriptor, and to modify the state of the associated description, other than through the **closedir**, **readdir**, **readdir_r**, or **rewinddir** subroutines, the behavior is undefined.

Return Values

If successful, the **dirfd** subroutine returns an integer that contains a file descriptor for the stream pointed to by *dirp* argument. Otherwise, the **dirfd** subroutine returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

The **dirfd** subroutine might fail if the following is true:

Item	Description
EINVAL	The *dirp* argument does not refer to a valid directory stream.
ENOTSUP	The implementation does not support the association of a file descriptor with a directory.

doupdate, refresh, wnoutrefresh, or wrefresh Subroutines

Purpose

Refreshes windows and lines.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int doupdate(void);

int refresh(void);

int wnoutrefresh(WINDOW  *win);

int wrefresh(WINDOW *win);
```

Description

The **refresh** and **wrefresh** subroutines refresh the current or specified window. The subroutines position the terminal's cursor at the cursor position of the window, except that, if the leaveok mode has been enabled, they may leave the cursor at an arbitrary position.

The **wnoutrefresh** subroutine determines which parts of the terminal may need updating.

The **doupdate** subroutine sends to the terminal the commands to perform any required changes.

Parameters

Item	Description
**win*	Specifies the window to be refreshed.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

For the **doupdate** or **wnoutrefresh** subroutine:

To update the user-defined windows `my_window1` and `my_window2`, enter:

```
WINDOW *my_window1, my_window2;
wnoutrefresh(my_window1);
wnoutrefresh(my_window2);
doupdate();
```

For the **refresh** or **wrefresh** subroutine:

1. To update the terminal's display and the current screen structure to reflect changes made to the standard screen structure, use:

   ```
   refresh();
   ```

2. To update the terminal and the current screen structure to reflect changes made to a user-defined window called `my_window`, use:

   ```
   WINDOW *my_window;
   wrefresh(my_window);
   ```

3. To restore the terminal to its state at the last refresh, use:

   ```
   wrefresh(curscr);
   ```

This subroutine is useful if the terminal becomes garbled for any reason.

drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, or srand48 Subroutine

Purpose

Generate uniformly distributed pseudo-random number sequences.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
double drand48 (void)
```

```
double erand48 ( xsubi)
unsigned short int xsubi[3];
```

```
long int jrand48 (xsubi)
unsigned short int xsubi[3];
```

```
void lcong48 ( Parameter)
unsigned short int Parameter[7];
```

```
long int lrand48 (void)
```

```
long int mrand48 (void)
```

```
long int nrand48 (xsubi)
unsigned short int xsubi[3];
```

```
unsigned short int *seed48 ( Seed16v)
unsigned short int Seed16v[3];
```

```
void srand48 ( SeedValue)
long int SeedValue;
```

Description

> **Attention:** Do not use the **drand48**, **erand48**, **jrand48**, **lcong48**, **lrand48**, **mrand48**, **nrand48**, **seed48**, or **srand48** subroutine in a multithreaded environment.

This family of subroutines generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The **drand48** subroutine and the **erand48** subroutine return positive double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

The **lrand48** subroutine and the **nrand48** subroutine return positive long integers uniformly distributed over the interval [0,2**31).

The **mrand48** subroutine and the **jrand48** subroutine return signed long integers uniformly distributed over the interval [-2**31, 2**31).

The **srand48** subroutine, **seed48** subroutine, and **lcong48** subroutine initialize the random-number generator. Programs must call one of them before calling the **drand48**, **lrand48** or **mrand48** subroutines. (Although it is not recommended, constant default initializer values are supplied if the **drand48**, **lrand48** or **mrand48** subroutines are called without first calling an initialization subroutine.) The **erand48**, **nrand48**, and **jrand48** subroutines do not require that an initialization subroutine be called first.

The previous value pointed to by the **seed48** subroutine is stored in a 48-bit internal buffer, and a pointer to the buffer is returned by the **seed48** subroutine. This pointer can be ignored if it is not needed, or it can be used to allow a program to restart from a given point at a later time. In this case, the pointer is accessed to retrieve and store the last value pointed to by the **seed48** subroutine, and this value is then used to reinitialize, by means of the **seed48** subroutine, when the program is restarted.

All the subroutines work by generating a sequence of 48-bit integer values, *x*[*i*], according to the linear congruential formula:

```
x[n+1] = (ax[n] + c)mod m, n is > = 0
```

The parameter *m* = 248; hence 48-bit integer arithmetic is performed. Unless the **lcong48** subroutine has been called, the multiplier value *a* and the addend value *c* are:

```
a = 5DEECE66D base 16 = 273673163155 base 8
```

```
c = B base 16 = 13 base 8
```

Parameters

Item	Description
xsubi	Specifies an array of three shorts, which, when concatenated together, form a 48-bit integer.
SeedValue	Specifies the initialization value to begin randomization. Changing this value changes the randomization pattern.
Seed16v	Specifies another seed value; an array of three unsigned shorts that form a 48-bit seed value.
Parameter	Specifies an array of seven shorts, which specifies the initial *xsubi* value, the multiplier value *a* and the add-in value *c*.

Return Values

The value returned by the **drand48**, **erand48**, **jrand48, lrand48**, **nrand48**, and **mrand48** subroutines is computed by first generating the next 48-bit *x*[*i*] in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (most significant) bits of *x*[*i*] and transformed into the returned value.

The **drand48**, **lrand48**, and **mrand48** subroutines store the last 48-bit *x*[*i*] generated into an internal buffer; this is why they must be initialized prior to being invoked.

The **erand48**, **jrand48**, and **nrand48** subroutines require the calling program to provide storage for the successive *x*[*i*] values in the array pointed to by the *xsubi* parameter. This is why these routines do not have to be initialized; the calling program places the desired initial value of *x*[*i*] into the array and pass it as a parameter.

By using different parameters, the **erand48**, **jrand48**, and **nrand48** subroutines allow separate modules of a large program to generate independent sequences of pseudo-random numbers. In other words, the sequence of numbers that one module generates does not depend upon how many times the subroutines are called by other modules.

The **lcong48** subroutine specifies the initial *x*[*i*] value, the multiplier value *a*, and the addend value *c*. The *Parameter* array elements *Parameter*[0-2] specify *x*[*i*], *Parameter*[3-5] specify the multiplier *a*, and *Parameter*[6] specifies the 16-bit addend *c*. After **lcong48** has been called, a subsequent call to either the **srand48** or **seed48** subroutine restores the standard *a* and *c* specified before.

The initializer subroutine **seed48** sets the value of *x*[*i*] to the 48-bit value specified in the array pointed to by the *Seed16v* parameter. In addition, **seed48** returns a pointer to a 48-bit internal buffer that contains the previous value of *x*[*i*] that is used only by **seed48**. The returned pointer allows you to restart the pseudo-random sequence at a given point. Use the pointer to copy the previous *x*[*i*] value into a temporary array. Then call **seed48** with a pointer to this array to resume processing where the original sequence stopped.

The initializer subroutine **srand48** sets the high-order 32 bits of *x*[*i*] to the 32 bits contained in its parameter. The low order 16 bits of *x*[*i*] are set to the arbitrary value 330E16.

drem Subroutine

Purpose

Computes the IEEE Remainder as defined in the IEEE Floating-Point Standard.

Libraries

IEEE Math Library (**libm.a**) or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
```

double drem (*x*, *y*)
double *x*, *y*;

Description

The **drem** subroutine calculates the remainder r equal to x minus n to the x power multiplied by y (*r = x - n* * y), where the n parameter is the integer nearest the exact value of x divided by y (x/y). If |n -x/y| = 1/2, then the *n* parameter is an even value. Therefore, the remainder is computed exactly, and the absolute value of r (|r|) is less than or equal to the absolute value of y divided by 2 (|y|/2).

The IEEE Remainder differs from the **fmod** subroutine in that the IEEE Remainder always returns an *r* parameter such that |r| is less than or equal to |y|/2, while FMOD returns an *r* such that |r| is less than or equal to |y|. The IEEE Remainder is useful for argument reduction for transcendental functions.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **-lm** flag. For example: compile the **drem.c** file:

```
cc drem.c -lm
```

Note: For new development, the **remainder** subroutine is the preferred interface.

Parameters

Item	Description
x	Specifies double-precision floating-point value.
y	Specifies a double-precision floating-point value.

Return Values

The **drem** subroutine returns a NaNQ value for (x, 0) and (+/-INF, y).

drw_lock_done Kernel Service

Purpose

Unlock a disabled read-write lock.

Syntax

```
#include <sys/lock_def.h>
```

void drw_lock_done(*lock_addr***)**
drw_lock_t *lock_addr* **;**

Parameters

Item	Description
lock_addr	Specifies the address of the lock word to unlock.

Description

The **drw_lock_done** service unlocks the specified read-write lock. The calling thread or interrupt handler must own the lock either in read shared or write exclusive mode. The **drw_lock_done** service has no return values.

Execution Environment

The **drw_lock_done** kernel service may be called from either the process environment or the interrupt environment. However, if called from the process environment, interrupts must be disabled to some interrupt priority other than **INTBASE**.

Return Values

Done

drw_lock_free Kernel Service

Purpose

Frees resources associated with a disabled read-write lock.

Syntax

```
#include <sys/lock_def.h>
```

void drw_lock_free(*lock_addr***)**
drw_lock_t *lock_addr* **;**

Parameters

Item	Description
lock_addr	Specifies the address of the lock word to free.

Description

The **drw_lock_free** service frees the specified read-write lock and all internal resources that might be associated with the lock.

Execution Environment

The **drw_lock_free()** kernel service may be called from either the process environment or the interrupt environment.

Return Values

None

drw_lock_init Kernel Service

Purpose

Initialize a disabled read-write lock.

Syntax

```
#include <sys/lock_def.h>
```

void drw_lock_init(*lock_addr*)
drw_lock_t *lock_addr* **;**

Parameters

Item	Description
lock_addr	Specifies the address of the lock word to initialize.

Description

The **drw_lock_init** service initializes the specified read-write lock. The **drw_lock_init** service has no return values.

Execution Environment

The **drw_lock_init()** kernel service must be called from the process environment only.

Return Values

None

drw_lock_islocked Kernel Service

Purpose

Determine whether a **drw_lock** is held in either read or write mode.

Syntax

```
#include <sys/lock_def.h>
```

boolean_t drw_lock_islocked (*lock_addr*)
)drw_lock_t *lock_addr* **;**

Parameters

Item	Description
lock_addr	Specifies the address of the lock word.

Description

The **drw_lock_islocked** kernel services returns FALSE if the specified lock is not held in read or write mode. It returns TRUE if the lock is locked at the time of the call.

Execution Environment

The **drw_lock_islocked** kernel service may be called from either the process environment or the interrupt environment. However, if called from the process environment, interrupts must be disabled to some interrupt priority other than **INTBASE**.

Return Values

The following only apply to drw_lock_read_to_write:

Return value	Description
TRUE	Indicates that the lock is not currently held.
FALSE	Indicates that the lock is held.

drw_lock_read Kernel Service

Purpose

Lock a disabled read-write lock in read-shared mode.

Syntax

```
#include <sys/lock_def.h>
```

void drw_lock_read(*lock_addr***)**
drw_lock_t *lock_addr* **;**

Parameters

Item	Description
lock_addr	Specifies the address of the lock word to lock.

Description

The **drw_lock_read** service locks the specified read-write lock in read shared mode. The lock must have been previously initialized with the **lock_init** kernel service. The **drw_lock_read** service has no return values.

Execution Environment

The **drw_lock_read** kernel service may be called from either the process environment or the interrupt environment. However, if called from the process environment, interrupts must be disabled to some interrupt priority other than **INTBASE**.

Return Values

None

drw_lock_read_to_write Kernel Service

Purpose

Upgrades a disabled read-write from read-shared to write exclusive mode.

Syntax

```
#include <sys/lock_def.h>
```

boolean drw_lock read_to_write (*lock_addr***)**
boolean drw_lock try_read_to_write (*lock_addr***)drw_lock_t** *lock_addr* **;**

Parameters

Item	Description
lock_addr	Specifies the address of the lock word to lock.

Description

The **drw_lock_read_to_write** and **drw_lock_try_read_to_write** kernel services try to upgrade the specified read-write lock from read-shared to write-exclusive mode. The caller must hold the lock in read mode. The lock is successfully upgraded if no other thread has already requested write-exclusive access for this lock. If the lock cannot be upgraded, it is no longer held on return from the **drw_lock_read_to_write** kernel service; it is still held in shared-read mode on return from the **drw_lock_try_read_to_write** kernel service.

The calling kernel thread must hold the lock in shared-read mode.

Execution Environment

The **drw_lock_read_to_write** and **drw_lock_try_read_to_write** kernel services may be called from either the process environment or the interrupt environment. However, if called from the process environment, interrupts must be disabled to some interrupt priority other than INTBASE.

Return Values

The following only apply to **drw_lock_read_to_write**:

Item	Description
TRUE	Indicates that the lock was successfully upgraded to exclusive-write mode.
FALSE	Indicates that the lock was not upgraded to exclusive-write mode and the lock is no longer held by the caller.

The following only apply to **lock_try_read_to_write**:

Item	Description
TRUE	Indicates that the lock was successfully upgraded to exclusive-write mode.
FALSE	Indicates that the lock was not upgraded and is held in read mode.

drw_lock_try_write Kernel Service

Purpose

Immediately acquire a disabled read-write lock in write-exclusive mode if available.

Syntax

```
#include <sys/lock_def.h>
```

boolean_t drw_lock try_write (*lock_addr***)**
drw_lock_t *lock_addr* **;**

Parameters

lock_addr
Specifies the address of the lock word to lock.

Description

The **drw_lock_try_write** kernel service acquires an available **drw_lock** in write mode and returns TRUE. It returns FALSE if the lock is not available.

Execution Environment

The **drw_lock_try_write** kernel service may be called from either the process environment or the interrupt environment. However, if called from the process environment, interrupts must be disabled to some interrupt priority other than **INTBASE**.

Return Values

The following only apply to **drw_lock_try_write**:

TRUE
Indicates that the lock was acquired.

FALSE
Indicates that the lock was not acquired.

drw_lock_write Kernel Service

Purpose

Lock a disabled read-write lock in write-exclusive mode.

Syntax

```
#include <sys/lock_def.h>
```

void drw_lock_write(*lock_addr***)**
drw_lock_t *lock_addr* **;**

Parameters

Item	Description
lock_addr	Specifies the address of the lock word to lock.

Description

The **drw_lock_write** service locks the specified read-write lock in write-exclusive mode. The lock must have been previously initialized with the **lock_init** kernel service. The **drw_lock_write** service has no return values.

Execution Environment

The **drw_lock_write** kernel service may be called from either the process environment or the interrupt environment. However, if called from the process environment, interrupts must be disabled to some interrupt priority other than **INTBASE**.

Return Values

None

drw_lock_write_to_read Kernel Service

Purpose

Downgrades a disabled read-write lock from write exclusive mode to read-shared mode.

Syntax

```
#include <sys/lock_def.h>
```

void drw_lock_write_to_read(*lock_addr***)**
drw_lock_t *lock_addr* **;**

Parameters

Item	Description
lock_addr	Specifies the address of the lock word to lock.

Description

The **drw_lock_write_to_read** kernel service downgrades the specified complex lock from exclusive-write mode to shared-read mode. The calling kernel thread must hold the lock in exclusive-write mode.

Once the lock has been downgraded to shared-read mode, other kernel threads will also be able to acquire it in read-shared mode.

Execution Environment

The **drw_lock_write_to_read** kernel service may be called from either the process environment or the interrupt environment. However, if called from the process environment, interrupts must be disabled to some interrupt priority other than **INTBASE**.

Return Values

None

dscr_ctl Subroutine

Purpose

Allows applications to read the current settings of the hardware streams mechanism and to set the system-wide or per-process values for the Data Streams Control Register (DSCR).

Note: The DSCR is privileged. It can be read or written only by the operating system. Beginning with POWER8, per-thread problem-state (user) access to the DSCR is allowed through Special Purpose Register (SPR) 3, as defined by the PowerISA.

Syntax

```
#include <sys/machine.h>
```

```
int dscr_ctl(int operation, void * buf_p, int size);
```

Description

The DSCR register consists of several bit fields:

Bit Position	Name	Description
39	SWTE (Software Transient Enable)	Applies the transient attribute to software defined streams.
40	HWTE (Hardware Transient Enable)	Applies the transient attribute to hardware detected streams.
41	STE (Store Transient Enable)	Applies the transient attribute to store streams.
42	LTE (Load Transient Enable)	Applies the transient attribute to load streams.
43	SWUE (Software Unit Count Enable)	Applies the unit count to software defined streams.
44	HWUE (Hardware Unit Count Enable)	Applies the unit count to hardware defined streams.
45-54	UNITCNT (Unit Count)	Number of units in a data stream.
55-57	URG (Depth Attainment Urgency)	Indicates the time of prefetch depth that can be reached for the hardware-detected streams.
58	LSD (Load Stream Disable)	Disables the hardware detection and the initiation of load streams.
59	SNSE (Stride-N Stream Enable)	Enables the hardware detection and initiation of load and store streams that have a stride greater than a single cache block. The load streams are detected only when the **LSD** bit is zero. The store streams are detected only when the **SSE** bit is one.
60	SSE (Store Stream Enable)	Enables the hardware detection and the initiation of store streams.
61-63	DPFD (Default Prefetch Depth)	Applies the depth value for the hardware-detected streams and software-defined streams for which a **dcbt** instruction with the TH value as 1010 is not used.

The firmware provides a platform default value for the DSCR register. When the prefetch depth is set to **0** in the DSCR register, the processor uses this default value implicitly.

The **dscr_ctl** system call allows a privileged application to set an operating system default value for the DSCR, which overrides the platform default.

The **dscr_ctl** system call allows any application to set a per-process value for the DSCR register, which overrides the operating system default value for this process.

When a thread issues the **dscr_ctl** system call to change the prefetch depth for the process, the new value is written into the AIX process context and the DSCR of the thread that runs the system call. If another

thread in the process is simultaneously running on another processor, it starts using the new DSCR value only after the new value is reloaded from the process context.

When a thread starts running on a processor, the value of the DSCR for the owning process is written in the DSCR register. If the process has not set its DSCR value with the **dscr_ctl** system call, the operating system default value is used.

When the **fork** subroutine is called, the new process inherits the DSCR value from its parent process. This value gets reset to the system default value when the **exec** subroutine is called.

On systems which support programmatic setting of the DSCR through problem-state (user) access, such as POWER8, the value set by such access is thread-specific and overrides any other values, even the ones that are written through this service. In other words, problem-state manipulation of the DSCR provides for the finest granularity of access (per-thread) to the hardware streams functionality.

The following symbolic values for the various fields are defined in the <sys/machine.h> file:

```
DPFD_DEFAULT            0
DPFD_NONE               1
DPFD_SHALLOWEST         2
DPFD_SHALLOW            3
DPFD_MEDIUM             4
DPFD_DEEP               5
DPFD_DEEPER             6
DPFD_DEEPEST            7

DSCR_SSE                1<<3
DSCR_SNSE               1<<4
DSCR_LSD                1<<5

URG_DEFAULT             0<<6
URG_NOT_URGENT          1<<6
URG_LEAST_URGENT        2<<6
URG_LEAST_URGENT        3<<6
URG_LESS_URGENT         4<<6
URG_MEDIUM              5<<6
URG_MORE_URGENT         6<<6
URG_MOST_URGENT         7<<6
```

```
DSCR_HWUE (1<<19)
DSCR_SWUE (1<<20)
DSCR_LTE  (1<<21)
DSCR_STE  (1<<22)
DSCR_HWTE (1<<23)
DSCR_SWTE (1<<24)
```

The following is the description of the **dscr_properties** structure in the <sys/machine.h> file:

```
struct dscr_properties {
      uint        version;             /* Properties struct version                  */
      uint        number_of_streams;   /* Number of hardware streams                 */
      long long   platform_default_pd; /* PFW default DSCR value                     */
      long long   os_default_pd;       /* AIX default DSCR value                     */
      int         dscr_version;        /* Architecture version, such as PowerISA 2.07 */
      uint        dscr_control;        /* System-wide DSCR control (read only)       */
      long long   dscr_smt[5];         /* DSCR/SMT Matrix                            */
      long long   dscr_mask;           /* Mask of valid bits per architecture version */
};
```

Depending on the version of the Instruction Set Architecture (ISA) for Power Systems servers supported by a specific AIX level on a specified hardware platform, only a subset of the bits previously shown might be supported.

Refer to the <sys/machine.h> header file for the definitions for the **dscr_version** field and the corresponding bits supported for each version.

The following is the sample code setting of the DSCR value of the process:

```
#include <sys/machine.h>
int rc;
long long dscr = DSCR_SSE | DPFD_DEEPER;
rc = dscr_ctl(DSCR_WRITE, &dscr);
```

Parameters

Parameter	Description
Operation	Specifies the operation to perform. It has the following flags: **DSCR_WRITE** Stores the new value from the input buffer into the process context and in the DSCR. **DSCR_READ** Reads the current value of the DSCR and returns it to the output buffer. **DSCR_GET_PROPERTIES** Reads the number of hardware streams supported by the platform, the platform default prefetch depth used by the firmware, the operating system default prefetch depth, and the supported version of the ISA for Power Systems servers from the kernel memory. It returns values in the output buffer (struct `dscr_properties` defined in the `sys/machine.h` file). **DSCR_SET_DEFAULT** Sets the 64-bit DSCR value in the buffer that is pointed to by the **buf_p** parameter as the operating system default. Returns the previous default value in the buffer that is pointed to by the **buf_p** parameter. It requires the root authority. The new default value is used by all the processes that do not explicitly set a DSCR value by using the **DSCR_WRITE** flag. The new default value is not permanent across reboot operations. To permanently set the default prefetch depth for an operating system across reboot operations, use the **dscrtl** command.
buf_p	When this parameter is used with the **DSCR_WRITE**, **DSCR_READ** and **DSCR_GET_PROPERTIES** values, the **buf_p** parameter specifies the pointer to an area of memory, that is the input buffer from where the values are copied from or the output buffer to which the data is copied. The **buf_p** parameter must be a pointer to a 64-bit data area for the **DSCR_WRITE**, **DSCR_READ** and **DSCR_SET_DEFAULT** operations. The **buf_p** parameter must be a pointer to a struct **dscr_properties** defined in the `sys/machine.h` file for the **DSCR_GET_PROPERTIES** operation.
size	Specifies the size in bytes of the area pointed to by the **buf_p** parameter.

Return Values

Value	Description
0	Returns 0 when the **dscr_ctl** subroutine is successful.
-1	Returns -1 if an error is detected. In this case, **errno** is set to indicate the error.

Error Codes

When the **dscr_ctl** subroutine fails, **errno** is set to one of the following values:

Value of errno	Description
EFAULT	The address passed to the function is not valid.
EINVAL	The operation is **DSCR_WRITE** or **DSCR_SET_DEFAULT** and the value passed for DSCR is not valid.
ENOTSUP	Data streams are not supported by platform hardware.

Value of errno	Description
EPERM	Operation is not permitted. The **DSCR_SET_DEFAULT** operation is used by a nonroot user.

duplocale Subroutine

Purpose

Duplicates a locale object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <locale.h>
```

```
locale_t duplocale(locobj);
locale_t locobj;
```

Description

The **duplocale** subroutine creates a duplicate copy of the locale object that is referenced by the *locobj* argument.

If the *locobj* argument value is LC_GLOBAL_LOCALE, the **duplocale** subroutine creates a new locale object that contains a copy of the global locale that is determined by the **setlocale** subroutine.

If the *locobj* argument is not a valid handle for a locale object, the behavior of the **duplocale** subroutine is undefined.

Return Values

If successful, the **duplocale** subroutine returns a handle for a new locale object. Otherwise, the **duplocale** subroutine returns **(locale_t) 0** and sets the **errno** global variable to indicate the error.

Error Codes

The **duplocale** subroutine fails if the following is true:

Item	Description
ENOMEM	There is not enough memory available to create the locale object or load the locale data.

The **duplocale** subroutine might fail if the following is true:

Item	Description
EINVAL	The *locobj* argument is not a handle for a locale object.

e

The following Base Operating System (BOS) runtime services begin with the letter *e*.

_end, _etext, or _edata Identifier

Purpose

Define the first addresses following the program, initialized data, and all data.

Syntax

```
extern _end;
```

```
extern _etext;
```

```
extern _edata;
```

Description

The external names **_end**, **_etext**, and **_edata** are defined by the loader for all programs. They are not subroutines but identifiers associated with the following addresses:

Item	Description
_etext	The first address following the program text.
_edata	The first address following the initialized data region.
_end	The first address following the data region that is not initialized. The name **end** (with no underscore) defines the same address as does **_end** (with underscore).

The break value of the program is the first location beyond the data. When a program begins running, this location coincides with **end**. However, many factors can change the break value, including:

- The **brk** or **sbrk** subroutine
- The **malloc** subroutine
- The standard I/O subroutines
- The **-p** flag with the **cc** command

Therefore, use the **brk** or **sbrk(0)** subroutine, not the **end** address, to determine the break value of the program.

echo or noecho Subroutine

Purpose

Enables/disables terminal echo.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int echo(void);
```

```
int noecho(void);
```

Description

The **echo** subroutine enables Echo mode for the current screen. The **noecho** subroutine disables Echo mode for the current screen. Initially, curses software echo mode is enabled and hardware echo mode of the tty driver is disabled. The **echo** and **noecho** subroutines control software echo only. Hardware echo must remain disabled for the duration of the application, else the behaviour is undefined.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To turn echoing on, use:

   ```
   echo();
   ```

2. To turn echoing off, use:

   ```
   noecho();
   ```

echochar or wechochar Subroutines

Purpose

Echos single-byte character and rendition to a window and refreshes the window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int echochar(const chtype ch);
```

```
int wechochar(WINDOW *win,
const chtype ch);
```

Description

The **echochar** subroutine is equivalent to a call to the **addch** soubroutine followed by a call to the **refresh** subroutine.

The **wechochar** subroutine is equivalent to a call to the **waddch** subroutine followed by a call to the **wrefresh** subroutine.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Example

To output the character `I` to the stdscr at the present cursor location and to update the physical screen, do the following:

```
echochar('I');
```

ecvt, fcvt, or gcvt Subroutine

Purpose

Converts a floating-point number to a string.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
char *ecvt ( Value,  NumberOfDigits,  DecimalPointer,  Sign;)
double Value;
int  NumberOfDigits, *DecimalPointer, *Sign;
```

```
char *fcvt (Value, NumberOfDigits, DecimalPointer, Sign;)
double Value;
int  NumberOfDigits, *DecimalPointer, *Sign;
```

```
char *gcvt (Value, NumberOfDigits,  Buffer;)
double Value;
int NumberOfDigits;
char *Buffer;
```

Description

The **ecvt**, **fcvt**, and **gcvt** subroutines convert floating-point numbers to strings.

The **ecvt** subroutine converts the *Value* parameter to a null-terminated string and returns a pointer to it. The *NumberOfDigits* parameter specifies the number of digits in the string. The low-order digit is rounded according to the current rounding mode. The **ecvt** subroutine sets the integer pointed to by the *DecimalPointer* parameter to the position of the decimal point relative to the beginning of the string. (A negative number means the decimal point is to the left of the digits given in the string.) The decimal point itself is not included in the string. The **ecvt** subroutine also sets the integer pointed to by the *Sign* parameter to a nonzero value if the *Value* parameter is negative and sets a value of 0 otherwise.

The **fcvt** subroutine operates identically to the **ecvt** subroutine, except that the correct digit is rounded for C or FORTRAN F-format output of the number of digits specified by the *NumberOfDigits* parameter.

Note: In the F-format, the *NumberOfDigits* parameter is the number of digits desired after the decimal point. Large numbers produce a long string of digits before the decimal point, and then *NumberOfDigits* digits after the decimal point. Generally, the **gcvt** and **ecvt** subroutines are more useful for large numbers.

The **gcvt** subroutine converts the *Value* parameter to a null-terminated string, stores it in the array pointed to by the *Buffer* parameter, and then returns the *Buffer* parameter. The **gcvt** subroutine attempts

to produce a string of the *NumberOfDigits* parameter significant digits in FORTRAN F-format. If this is not possible, the E-format is used. The **gcvt** subroutine suppresses trailing zeros. The string is ready for printing, complete with minus sign, decimal point, or exponent, as appropriate. The radix character is determined by the current locale (see **setlocale** subroutine). If the **setlocale** subroutine has not been called successfully, the default locale, POSIX, is used. The default locale specifies a **.** (period) as the radix character. The **LC_NUMERIC** category determines the value of the radix character within the current locale.

The **ecvt**, **fcvt**, and **gcvt** subroutines represent the following special values that are specified in ANSI/IEEE standards 754-1985 and 854-1987 for floating-point arithmetic:

Item	Description
Quiet NaN	Indicates a quiet not-a-number (NaNQ)
Signalling NaN	Indicates a signaling NaNS
Infinity	Indicates a INF value

The sign associated with each of these values is stored in the *Sign* parameter.

Note: A value of 0 can be positive or negative. In the IEEE floating-point, zeros also have signs and set the *Sign* parameter appropriately.

Attention: All three subroutines store the strings in a static area of memory whose contents are overwritten each time one of the subroutines is called.

Parameters

Item	Description
Value	Specifies some double-precision floating-point value.
NumberOfDigits	Specifies the number of digits in the string.
DecimalPointer	Specifies the position of the decimal point relative to the beginning of the string.
Sign	Specifies that the sign associated with the return value is placed in the *Sign* parameter. In IEEE floating-point, since 0 can be signed, the *Sign* parameter is set appropriately for signed 0.
Buffer	Specifies a character array for the string.

efs_closeKS Subroutine

Purpose

Disassociates the processes with open keystores.

Library

EFS Library (**libefs.a**)

Syntax

```
#include <libefs.h>

int efs_closeKS(void)
```

Description

The **efs_closeKS** subroutine disassociates an open keystore with a process. Therefore, the process does not have access to the EFS keys and is not to encrypt or decrypt files. Opening an encrypted file produces the error **ENOATTR**.

If a keystore is open using the **efskeymgr** command or using the login process, the keys within the keystore are associated to user's process and child processes. These keys are used within an Encrypted File System (EFS) to encrypt and decrypt files. If the **efs_closeKS** subroutine is called, the process is disassociated with the keystores, and is no longer able to open, decrypt or read EFS files. The process is not be able to open, encrypt or write EFS files. If the process has previously opened EFS files, those file operations maintain the ability to encrypt and decrypt.

Return Values

If successful, the **efs_closeKS** subroutine returns a value of zero. If it fails, it returns a value of -1 and sets the **errno** error code.

Errors

No error code is defined.

Files

The**/etc/security/group** File and the **user** File in *Files Reference*.

EnableCriticalSections, BeginCriticalSection, and EndCriticalSection Subroutine

Purpose

Enables a thread to be exempted from timeslicing and signal suspension, and protects critical sections.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/thread_ctl.h>

int EnableCriticalSections(void);
void BeginCriticalSection(void);
void EndCriticalSection(void);
```

Description

When called, the **EnableCriticalSections** subroutine enables the thread to be exempted from timeslicing and signal suspension. Once that is done, the thread can call the **BeginCriticalSection** and **EndCriticalSection** subroutines to protect critical sections. Calling the **BeginCriticalSection** and **EndCriticalSection** subroutines with exemption disabled has no effect. The subroutines are safe for use by multithreaded applications.

Once the service is enabled, the thread can protect critical sections by calling the **BeginCriticalSection** and **EndCriticalSection** subroutines. Calling the **BeginCriticalSection** subroutine will exempt the thread from timeslicing and suspension. Calling the **EndCriticalSection** subroutine will clear exemption for the thread.

The **BeginCriticalSection** subroutine will not make a system call. The **EndCriticalSection** subroutine might make a system call if the thread was granted a benefit during the critical section. The purpose of the system call would be to notify the kernel that any posted but undelivered stop signals can be delivered, and any postponed timeslice can now be completed.

Return Values

The **EnableCriticalSections** subroutine returns a zero.

endwin Subroutine

Purpose

Suspends curses session.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int endwin(void)
```

Description

The **endwin** subroutine restores the terminal after Curses activity by at least restoring the saved shell terminal mode, flushing any output to the terminal and moving the cursor to the first column of the last line of the screen. Refreshing a window resumes program mode. The application must call the **endwin** subroutine for each terminal being used before exiting. If the **newterm** subroutine is called more than once for the same terminal, the first screen created must be the last one for which the **endwin** subroutine is called.

Return Values

Upon successful completion, the **endwin** subroutine returns OK. Otherwise, it returns ERR.

Examples

To terminate curses permanently or temporarily, enter:

```
endwin();
```

erase or werase Subroutine

Purpose

Copies blank spaces to every position in a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
erase( )
```

werase(*Window*)
WINDOW ***Window**;

Description

The **erase** and **werase** subroutines copy blank spaces to every position in the specified window. Use the **erase** subroutine with the stdscr and the **werase** subroutine with user-defined windows.

Parameters

Item	Description
Window	Specifies the window to erase.

Examples

1. To erase the standard screen structure, enter:

   ```
   erase();
   ```

2. To erase the user-defined window `my_window`, enter:

   ```
   WINDOW *my_window;
   werase(my_window);
   ```

erasechar, erasewchar, killchar, and killwchar Subroutine

Purpose

Terminal environment query functions.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

char erasechar(void);

int erasewchar(wchar_t *ch);

char killchar(void);
```

```
int killwchar(wchar_t
*ch);
```

Description

The **erasechar** subroutine returns the current character. chosen by the user. The **erasechar** subroutine stores the current erase character in the object pointed to by the *ch* parameter. If no erase character has been defined, the subroutine will fail and the object pointed to by *ch* will not be changed.

The **killchar** subroutine returns the current line.

The **killchar** subroutine stores the current line kill character in the object pointed to by *ch*. If no line kill character has been defined, the subroutine will fail and the object pointed to by *ch* will not be changed.

Return Values

The **erasechar** subroutine returns the erase character and the **killchar** subroutine returns the line kill character. The return value is unspecified when these characters are multi-byte characters.

Upon successful completion, the **erasechar** subroutine and the **killchar** subroutine return OK. Otherwise, they return ERR.

Examples

To retrieve a user's erase character and return it to the user-defined variable `myerase`, enter:

```
myerase = erasechar();
```

erf, erff, erfl, erfd32, erfd64, and erfd128 Subroutines

Purpose

Computes the error and complementary error functions.

Libraries

IEEE Math Library (**libm.a**) or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
```

double erf (*x*)
double *x*;

```
float erff (x)
float x;
```

```
long double erfl (x)
long double x;
_Decimal32 erfd32 (x)
_Decimal32 x;

_Decimal64 erfd64 (x)
_Decimal64 x;

_Decimal128 erfd128 (x)
_Decimal128 x;
```

Description

The **erf**, **erff**, **erfl**, **erfd32**, **erfd64**, and **erfd128** subroutines return the error function of the *x* parameter, defined for the **erf** subroutine as the following:

```
erf(x) = (2/sqrt(pi) * (integral [0 to x] of exp(-(t**2)) dt)
```

```
erfc(x) = 1.0 - erf(x)
```

Note: Compile any routine that uses subroutines from the **libm.a** library with the **-lm** flag. To compile the **erf.c** file, for example, enter:

```
cc erf.c -lm
```

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies a double-precision floating-point value.

Return Values

Upon successful completion, the **erf**, **erff**, **erfl**, **erfd32**, **erfd64**, and **erfd128** subroutines return the value of the error function.

If *x* is NaN, a NaN is returned.

If *x* is ±0, ±0 is returned.

If *x* is ±Inf, ±1 is returned.

If *x* is subnormal, a range error may occur, and 2 * *x*/**sqrt**(pi) should be returned.

erfc, erfcf, erfcl, erfcd32, erfcd64, and erfcd128 Subroutines

Purpose

Computes the complementary error function.

Syntax

```
#include <math.h>

float erfcf (x)
float x;

long double erfcl (x)
long double x;

double erfc (x)
double x;
_Decimal32 erfcd32 (x)
_Decimal32 x;
_Decimal64 erfcd64 (x)
_Decimal64 x;

_Decimal128 erfcd128 (x)
_Decimal128 x;
```

Description

The **erfcf**, **erfcl**, **erfc**, **erfcd32**, **erfcd64**, and **erfcd128** subroutines compute the complementary error function 1.0 **-** **erf**(*x*).

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. Upon return, if **errno** is nonzero or

fetestexcept(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **erfcf**, **erfcl**, **erfc**, **erfcd32**, **erfcd64**, and **erfcd128** subroutines return the value of the complementary error function.

If the correct value would cause underflow and is not representable, a range error may occur. Either 0.0 (if representable), or an implementation-defined value is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0, +1 is returned.

If *x* is -Inf, +2 is returned.

If *x* is +Inf, +0 is returned.

If the correct value would cause underflow and is representable, a range error may occur and the correct value is returned.

errlog Subroutine

Purpose

Logs an application error to the system error log.

Library

Run-Time Services Library (**librts.a**)

Syntax

```
#include <sys/errids.h>
int errlog ( ErrorStructure,  Length)
void *ErrorStructure;
unsigned int Length;
```

Description

The **errlog** subroutine writes an error log entry to the **/dev/error** file. The **errlog** subroutine is used by application programs.

The transfer from the **err_rec** structure to the error log is by a **write** subroutine to the **/dev/error** special file.

The **errdemon** process reads from the **/dev/error** file and writes the error log entry to the system error log. The timestamp, machine ID, node ID, and Software Vital Product Data associated with the resource name (if any) are added to the entry before going to the log.

Parameters

Item	Description

ErrorStructure	Points to an error record structure containing an error record. Valid error record structures are typed in the **/usr/include/sys/err_rec.h** file. The two error record structures available are **err_rec** and **err_rec0**. The **err_rec** structure is used when the `detail_data` field is required. When the `detail_data` field is not required, the **err_rec0** structure is used.
	
	`struct err_rec0 {`
	`  unsigned int  error_id;`
	`  char  resource_name[ERR_NAMESIZE];`
	`};`
	`struct err_rec {`
	`  unsigned int error_id;`
	`  char resource_name[ERR_NAMESIZE];`
	`  char  detail_data[1];`
	`};`
	
	The fields of the structures **err_rec** and **err_rec0** are:
	
	`error_id`
	Specifies an index for the system error template database, and is assigned by the **errupdate** command when adding an error template. Use the **errupdate** command with the **-h** flag to get a #`define` statement for this 8-digit hexadecimal index.
	
	`resource_name`
	Specifies the name of the resource that has detected the error. For software errors, this is the name of a software component or an executable program. For hardware errors, this is the name of a device or system component. It does not indicate that the component is faulty or needs replacement instead, it is used to determine the appropriate diagnostic modules to be used to analyze the error.
	
	`detail_data`
	Specifies an array from 0 to **ERR_REC_MAX** bytes of user-supplied data. This data may be displayed by the **errpt** command in hexadecimal, alphanumeric, or binary form, according to the `data_encoding` fields in the error log template for this `error_id` field.

Length	Specifies the length in bytes of the **err_rec** structure, which is equal to the size of the `error_id` and `resource_name` fields plus the length in bytes of the `detail_data` field.

Return Values

Item	Description
0	The entry was logged successfully.
-1	The entry was not logged.

Files

Item	Description
/dev/error	Provides standard device driver interfaces required by the error log component.

Item	Description
/usr/include/sys/errids.h	Contains definitions for error IDs.
/usr/include/sys/err_rec.h	Contains structures defined as arguments to the **errsave** kernel service and the **errlog** subroutine.
/var/adm/ras/errlog	Maintains the system error log.

errlog_close Subroutine

Purpose

Closes an open error log file.

Syntax

```
library liberrlog.a

#include <sys/errlog.h>

int errlog_close(handle)
errlog_handle_t handle;
```

Description

The error log specified by the handle argument is closed. The handle must have been returned from a previous **errlog_open** call.

Return Values

Upon successful completion, the **errlog_close** subroutine returns 0.

If an error occurs, the **errlog_close** subroutine returns **LE_ERR_INVARG**.

errlog_find_first, errlog_find_next, and errlog_find_sequence Subroutines

Purpose

Retrieves an error log entry using supplied criteria.

Syntax

```
library liberrlog.a

#include <sys/errlog.h>

int errlog_find_first(handle, filter, result)
errlog_handle_t handle;
errlog_match_t *filter;
errlog_entry_t *result;

int errlog_find_next(handle, result)
errlog_handle_t handle;
errlog_entry_t *result;

int errlog_find_sequence(handle, sequence, result)
errlog_handle_t handle;
int sequence;
errlog_entry_t *result;
```

Description

The **errlog_find_first** subroutine finds the first occurrence of the search argument specified by filter using the direction specified by the **errlog_set_direction** subroutine. The reverse direction is used if none was specified. In other words, by default, entries are searched starting with the most recent entry.

The **errlog_match_t** structure, pointed to by the filter parameter, defines a test expression or set of expressions to be applied to each errlog entry.

If the value passed in the filter parameter is null, the **errlog_find_first** subroutine returns the first entry in the log, and the **errlog_find_next** subroutine can then be used to return subsequent entries. To read all log entries in the desired direction, open the log, then issue **errlog_find_next** calls.

To define a basic expression, **em_field** must be set to the field in the errlog entry to be tested, **em_op** must be set to the relational operator to be applied to that field, and either **em_intvalue** or **em_strvalue** must be set to the value to test against. Basic expressions may be combined by attaching them to **em_left** and **em_right** of another **errlog_match_t** structure and setting **em_op** of that structure to a binary or unary operator. These complex expressions may then be combined with other basic or complex expressions in the same fashion to build a tree that can define a filter of arbitrary complexity.

The **errlog_find_next** subroutine finds the next error log entry matching the criteria specified by a previous **errlog_find_first** call. The search continues in the direction specified by the **errlog_set_direction** subroutine or the reverse direction by default.

The **errlog_find_sequence** subroutine returns the entry matching the specified error log sequence number, found in the **el_sequence** field of the **errlog_entry** structure.

Parameters

The handle contains the handle returned by a prior call to errlog_open.

The filter parameter points to an **errlog_match_t** element defining the search argument, or the first of an argument tree.

The sequence parameter contains the sequence number of the entry to be retrieved.

The result parameter must point to the area to contain the returned error log entry.

Return Values

Upon successful completion, the **errlog_find_first**, **errlog_find_next**, and **errlog_find_sequence** subroutines return 0, and the memory referenced by result contains the found entry.

The following errors may be returned:

Item	Description
LE_ERR_INVARG	A parameter error was detected.
LE_ERR_NOMEM	Memory could not be allocated.
LE_ERR_IO	An i/o error occurred.
LE_ERR_DONE	No more entries were found.

Examples

The code below demonstrates how to search for all errlog entries in a date range and with a class of **H** (hardware) or **S** (software).

```
{
    extern int        begintime, endtime;

    errlog_match_t    beginstamp, endstamp, andstamp;
    errlog_match_t    hardclass, softclass, orclass;
    errlog_match_t    andtop;
    int               ret;
```

```
    errlog_entry_t    result;

    /*
     *  Select begin and end times
     */
    beginstamp.em_op = LE_OP_GT;                /* Expression 'A' */
    beginstamp.em_field = LE_MATCH_TIMESTAMP;
    beginstamp.em_intvalue=begintime;

    endstamp.em_op = LE_OP_LT;                  /* Expression 'B' */
    endstamp.em_field = LE_MATCH_TIMESTAMP;
    endstamp.em_intvalue=endtime;

    andstamp.em_op = LE_OP_AND;                 /* 'A' and 'B' */
    andstamp.em_left = &beginstamp;
    andstamp.em_right = &endstamp;

    /*
     * Select the classes we're interested in.
     */
    hardclass.em_op = LE_OP_EQUAL;                /* Expression 'C' */
    hardclass.em_field = LE_MATCH_CLASS;
    hardclass.em_strvalue = "H";

    softclass.em_op = LE_OP_EQUAL;                /* Expression 'D' */
    softclass.em_field = LE_MATCH_CLASS;
    softclass.em_strvalue = "S";

    orclass.em_op = LE_OP_OR;                  /* 'C' or 'D' */
    orclass.em_left = &hardclass;
    orclass.em_right = &softclass;

    andtop.em_op = LE_OP_AND;                  /* ('A' and 'B') and ('C' or 'D') */
    andtop.em_left = &andstamp;
    andtop.em_right = &orclass;

    ret = errlog_find_first(handle, &andtop, &result);
}
```

The **errlog_find_first** function will return the first entry matching filter. Successive calls to the **errlog_find_next** function will return successive entries that match the filter specified in the most recent call to the **errlog_find_first** function. When no more matching entries are found, the **errlog_find_first** and **errlog_find_next** functions will return the value **LE_ERR_DONE**.

errlog_open Subroutine

Purpose

Opens an error log and returns a handle for use with other **liberrlog.a** functions.

Syntax

```
library liberrlog.a

#include <fcntl.h>
#include <sys/errlog.h>

int errlog_open(path, mode, magic, handle)
char            *path;
int              mode;
unsigned int     magic;
errlog_handle_t *handle;
```

Description

The error log specified by the path argument will be opened using mode. The handle pointed to by the handle parameter must be used with subsequent operations.

Parameters

The path parameter specifies the path to the log file to be opened. If path is NULL, the default errlog file will be opened. The valid values for mode are the same as they are for the open system subroutine. They can be found in the **fcntl.h** files.

The **magic** argument takes the **LE_MAGIC** value, indicating which version of the **errlog_entry_t** structure this application was compiled with.

Return Values

Upon successful completion, the **errlog_open** subroutine returns a 0 and sets the memory pointed to by handle to a handle used by subsequent **liberrlog** operations.

Upon error, the **errlog_open** subroutine returns one of the following:

Item	Description
LE_ERR_INVARG	A parameter error was detected.
LE_ERR_NOFILE	The log file does not exist.
LE_ERR_NOMEM	Memory could not be allocated.
LE_ERR_IO	An i/o error occurred.
LE_ERR_INVFILE	The file is not a valid error log.

errlog_set_direction Subroutine

Purpose

Sets the direction for the error log find functions.

Syntax

```
library liberrlog.a

#include <sys/errlog.h>

int errlog_set_direction(handle, direction)
errlog_handle_t handle;
int direction;
```

Description

The **errlog_find_next** and **errlog_find_sequence** subroutines search the error log starting with the most recent log entry and going backward in time, by default. The **errlog_set_direction** subroutine is used to alter this direction.

Parameters

The handle parameter must contain a handle returned by a previous **errlog_open** call.

The direction parameter must be **LE_FORWARD** or **LE_REVERSE**. **LE_REVERSE** is the default if the **errlog_set_direction** subroutine is not used.

Return Values

Upon successful completion, the **errlog_set_direction** subroutine returns 0.

If a parameter is invalid, the **errlog_set_direction** subroutine returns **LE_ERR_INVARG**.

errlog_write Subroutine

Purpose

Changes the previously read error log entry.

Syntax

```
library liberrlog.a

#include <sys/errlog.h>

int errlog_write(handle, entry)
errlog_handle_t handle;
errlog_entry_t *entry;
```

Description

The **errlog_write** subroutine is used to update the most recently read log entry. Neither the length nor the sequence number of the entry may be changed. The entry is simply updated in place.

If the **errlog_write** subroutine is used in a multi-threaded application, the program should obtain a lock around the read/write pair to avoid conflict.

Parameters

The handle parameter must contain a handle returned by a previous **errlog_open** call.

The entry parameter must point to an entry returned by the previous error log find function.

Return Values

Upon successful completion, the **errlog_write** subroutine returns 0.

If a parameter is invalid, the **errlog_write** subroutine returns **LE_ERR_INVARG**.

The **errlog_write** subroutine may also return one of the following:

Item	Description
LE_ERR_INVFILE	The data on file is invalid.
LE_ERR_IO	An i/o error occurred.
LE_ERR_NOWRITE	The entry to be written didn't match the entry being updated.

exec, execl, execle, execlp, execv, execve, execvp, exect, or fexecve Subroutine

Purpose

Executes a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
extern
char **environ;
```

```
int execl (
 Path,
 Argument0 [, Argument1, ...], 0)
const char *Path, *Argument0, *Argument
1, ...;
```

```
int execle (
 Path,
 Argument0 [, Argument1, ...], 0,

 EnvironmentPointer)
const
char *Path, *Argument0, *Argum
ent
1, ...;
char *const EnvironmentPointer[ ];
```

```
int execlp (
 File,
 Argument0 [, Argument1
, ...], 0)
const char *File, *Argument0, *Argument
1, ...;
```

```
int execv (
 Path,
 ArgumentV)
const char *Path;
char *const ArgumentV[ ];
```

```
int execve (
 Path,
 ArgumentV,

 EnvironmentPointer)
const char *Path;
char
*const ArgumentV[ ], *EnvironmentPointer
[ ];
```

```
int execvp (
 File,
 ArgumentV)
const char *File;
char *const ArgumentV[ ];
```

```
int exect (
 Path,
 ArgumentV,
 EnvironmentPointer)
char *Path, *ArgumentV, *EnvironmentPointer [ ];
```

```
int fexecve (FileDescriptor, ArgumentV, EnvironmentPointer)
int FileDescriptor;
char *const ArgumentV[ ], *EnvironmentPointer[ ];
```

Description

The **exec** subroutine, in all its forms, executes a new program in the calling process. The **exec** subroutine does not create a new process, but overlays the current program with a new one, which is called the *new-process image*. The new-process image file can be one of three file types:

- An executable binary file in XCOFF file format.
- An executable text file that contains a shell procedure (only the **execlp** and **execvp** subroutines allow this type of new-process image file).
- A file that names an executable binary file or shell procedure to be run.

The **fexecve** subroutine is equivalent to the **execve** subroutine, except that the fexecve subroutine takes the file descriptor of an open file to be executed as a first parameter, instead of a pathname. However, the following apply:

Note:

- If the file is a shell procedure that is deleted after the open operation, the **fexecve** subroutine starts the shell, but the shell cannot find the file.
- If the file is a shell procedure and the parent directory of the file is deleted after the file open operation, the **fexecve** subroutine returns an **ENOENT** error code.
- The **fexecve** subroutine does not check the Role Based Access Control (RBAC) execute permission.

The new-process image inherits the following attributes from the calling process image: session membership, supplementary group IDs, process signal mask, and pending signals.

The last of the types mentioned is recognized by a header with the following syntax:

```
#! Path [String]
```

The **#!** is the file *magic number*, which identifies the file type. The path name of the file to be executed is specified by the *Path* parameter. The *String* parameter is an optional character string that contains no tab or space characters. If specified, this string is passed to the new process as an argument in front of the name of the new-process image file. The header must be terminated with a new-line character. When called, the new process passes the *Path* parameter as *ArgumentV*[0]. If a *String* parameter is specified in the new process image file, the **exec** subroutine sets *ArgumentV*[0] to the *String* and *Path* parameter values concatenated together. The rest of the arguments passed are the same as those passed to the **exec** subroutine.

The **exec** subroutine attempts to cancel outstanding **asynchronous I/O requests** by this process. If the asynchronous I/O requests cannot be canceled, the application is blocked until the requests have completed.

The **exec** subroutine is similar to the **load** subroutine, except that the **exec** subroutine does not have an explicit library path parameter. Instead, the **exec** subroutine uses either the **LIBPATH** or LD_LIBRARY_PATH environment variable. The **LIBPATH** variable, when set, is used in favor of LD_LIBRARY_PATH; otherwise, LD_LIBRARY_PATH is used. These library path variables are ignored when the program that the **exec** subroutine is run on has more privilege than the calling program (for example, an **suid** program).

The **exect** subroutine is included for compatibility with older programs being traced with the **ptrace** command. The program being executed is forced into hardware single-step mode.

Note: exect is not supported in 64-bit mode.

Note: Currently, a Graphics Library program cannot be overlaid with another Graphics Library program. The overlaying program can be a nongraphics program. For additional information, see the **/usr/lpp/GL/README** file.

Parameters

Item	Description
Path	Specifies a pointer to the path name of the new-process image file. If Network File System (NFS) is installed on your system, this path can cross into another node. Data is copied into local virtual memory before proceeding.

Item	Description
File	Specifies a pointer to the name of the new-process image file. Unless the *File* parameter is a full path name, the path prefix for the file is obtained by searching the directories named in the **PATH** environment variable. The initial environment is supplied by the shell. **Note:** The **execlp** subroutine and the **execvp** subroutine take *File* parameters, but the rest of the **exec** subroutines take *Path* parameters. (For information about the environment, see the **environment** miscellaneous facility and the **sh** command.)
Argument0 [, *Argument1*, ...]	Point to null-terminated character strings. The strings constitute the argument list available to the new process. By convention, at least the *Argument0* parameter must be present, and it must point to a string that is the same as the *Path* parameter or its last component.
ArgumentV	Specifies an array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, the *ArgumentV* parameter must have at least one element, and it must point to a string that is the same as the *Path* parameter or its last component. The last element of the *ArgumentV* parameter is a null pointer.
EnvironmentPointer	An array of pointers to null-terminated character strings. These strings constitute the environment for the new process. The last element of the *EnvironmentPointer* parameter is a null pointer.
FileDescriptor	Specifies the file descriptor of an open file to be executed.

When a C program is run, it receives the following parameters:

```
main (ArgumentCount, ArgumentV, EnvironmentPointer)
int ArgumentCount;
char *ArgumentV[ ], *EnvironmentPointer[
];
```

In this example, the *ArgumentCount* parameter is the argument count, and the *ArgumentV* parameter is an array of character pointers to the arguments themselves. By convention, the value of the *ArgumentCount* parameter is at least 1, and the *ArgumentV*[0] parameter points to a string containing the name of the new-process image file.

The **main** routine of a C language program automatically begins with a runtime start-off routine. This routine sets the **environ** global variable so that it points to the environment array passed to the program in *EnvironmentPointer*. You can access this global variable by including the following declaration in your program:

```
extern char **environ;
```

The **execl**, **execv**, **execlp**, and **execvp** subroutines use the **environ** global variable to pass the calling process current environment to the new process.

File descriptors open in the calling process remain open, except for those whose **close-on-exec** flag is set. For those file descriptors that remain open, the file pointer is unchanged. (For information about file control, see the **fcntl.h** file.)

The state-of-conversion descriptors and message-catalog descriptors in the new process image are undefined. For the new process, an equivalent of the **setlocale** subroutine, specifying the **LC_ALL** value for its category and the **"C"** value for its locale, is run at startup.

If the new program requires shared libraries, the **exec** subroutine finds, opens, and loads each of them into the new-process address space. The referenced counts for shared libraries in use by the issuer of the **exec** are decremented. Shared libraries are searched for in the directories listed in the **LIBPATH** environment variable. If any of these files is remote, the data is copied into local virtual memory.

The **exec** subroutines reset all caught signals to the default action. Signals that cause the default action continue to do so after the **exec** subroutines. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset. (For information about signals, see the **sigaction** subroutine.)

If the *SetUserID* mode bit of the new-process image file is set, the **exec** subroutine sets the effective user ID of the new process to the owner ID of the new-process image file. Similarly, if the *SetGroupID* mode bit of the new-process image file is set, the effective group ID of the new process is set to the group ID of the new-process image file. The real user ID and real group ID of the new process remain the same as those of the calling process. (For information about the *SetID* modes, see the **chmod** subroutine.)

At the end of the **exec** operation the saved user ID and saved group ID of the process are always set to the effective user ID and effective group ID, respectively, of the process.

When one or both of the set ID mode bits is set and the file to be executed is a remote file, the file user and group IDs go through outbound translation at the server. Then they are transmitted to the client node where they are translated according to the inbound translation table. These translated IDs become the user and group IDs of the new process.

Note: setuid and **setgid** bids on shell scripts do not affect user or group IDs of the process finally executed.

Profiling is disabled for the new process.

The new process inherits the following attributes from the calling process:

- Nice value (see the **getpriority** subroutine, **setpriority** subroutine, **nice** subroutine)
- Process ID
- Parent process ID
- Process group ID
- **semadj** values (see the **semop** subroutine)
- tty group ID (see the **exit**, **atexit**, or **_exit** subroutine, **sigaction** subroutine)
- **trace** flag (see request 0 of the **ptrace** subroutine)
- Time left until an alarm clock signal (see the **incinterval** subroutine, **setitimer** subroutine, and **alarm** subroutine)
- Current directory
- Root directory
- File-mode creation mask (see the **umask** subroutine)
- File size limit (see the **ulimit** subroutine)
- Resource limits (see the **getrlimit** subroutine, **setrlimit** subroutine, and **vlimit** subroutine)
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_ctime` fields of the **tms** structure (see the **times** subroutine)
- Login user ID

Upon successful completion, the **exec** subroutines mark for update the `st_atime` field of the file.

Examples

1. To run a command and pass it a parameter, enter:

   ```
   execlp("ls", "ls", "-al", 0);
   ```

   The **execlp** subroutine searches each of the directories listed in the **PATH** environment variable for the **ls** command, and then it overlays the current process image with this command. The **execlp** subroutine is not returned, unless the **ls** command cannot be executed.

   Note: This example does not run the shell command processor, so operations interpreted by the shell, such as using wildcard characters in file names, are not valid.

2. To run the shell to interpret a command, enter:

   ```
   execl("/usr/bin/sh", "sh", "-c", "ls -l *.c",
   0);
   ```

   This runs the **sh** command with the **-c** flag, which indicates that the following parameter is the command to be interpreted. This example uses the **execl** subroutine instead of the **execlp** subroutine because the full path name **/usr/bin/sh** is specified, making a path search unnecessary.

   Running a shell command in a child process is generally more useful than simply using the **exec** subroutine, as shown in this example. The simplest way to do this is to use the **system** subroutine.

3. The following is an example of a new-process file that names a program to be run:

   ```
   #! /usr/bin/awk -f
   { for (i = NF; i > 0; --i) print $i }
   ```

   If this file is named `reverse`, entering the following command on the command line:

   ```
   reverse chapter1 chapter2
   ```

   This command runs the following command:

   ```
   /usr/bin/awk -f reverse chapter1 chapter2
   ```

   Note: The **exec** subroutines use only the first line of the new-process image file and ignore the rest of it. Also, the **awk** command interprets the text that follows a # (pound sign) as a comment.

Return Values

Upon successful completion, the **exec** subroutines do not return because the calling process image is overlaid by the new-process image. If the **exec** subroutines return to the calling process, the value of -1 is returned and the **errno** global variable is set to identify the error.

Error Codes

If the **exec** subroutine is unsuccessful, it returns one or more of the following error codes:

Item	Description
EACCES	The new-process image file is not an ordinary file.
EACCES	The mode of the new-process image file denies execution permission.
ENOEXEC	The **exec** subroutine is neither an **execlp** subroutine nor an **execvp** subroutine. The new-process image file has the appropriate access permission, but the magic number in its header is not valid.
ENOEXEC	The new-process image file has a valid magic number in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.

Item	Description
ETXTBSY	The new-process image file is a pure procedure (shared text) file that is currently open for writing by some process.
ENOMEM	The new process requires more memory than is allowed by the system-imposed maximum, the **MAXMEM** compiler option.
E2BIG	The number of bytes in the new-process argument list is greater than the system-imposed limit. This limit is a system configurable value that can be set by superusers or system group users using SMIT. Refer to **Kernel Tunable Parameters** for details.
EFAULT	The *Path*, *ArgumentV*, or *EnvironmentPointer* parameter points outside of the process address space.
EPERM	The *SetUserID* or *SetGroupID* mode bit is set on the process image file. The translation tables at the server or client do not allow translation of this user or group ID.

If the **exec** subroutine is unsuccessful because of a condition requiring path name resolution, it returns one or more of the following error codes:

Item	Description
EACCES	Search permission is denied on a component of the path prefix. Access could be denied due to a secure mount.
EFAULT	The *Path* parameter points outside of the allocated address space of the process.
EIO	An input/output (I/O) error occurred during the operation.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENAMETOOLONG	A component of a path name exceeded 255 characters and the process has the **disallow truncation** attribute (see the **ulimit** subroutine), or an entire path name exceeded 1023 characters.
ENOENT	A component of the path prefix does not exist.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENOENT	The path name is null.
ENOTDIR	A component of the path prefix is not a directory.
ESTALE	The root or current directory of the process is located in a virtual file system that has been unmounted.

In addition, some errors can occur when using the new-process file after the old process image has been overwritten. These errors include problems in setting up new data and stack registers, problems in mapping a shared library, or problems in reading the new-process file. Because returning to the calling process is not possible, the system sends the **SIGKILL** signal to the process when one of these errors occurs.

If an error occurred while mapping a shared library, an error message describing the reason for error is written to standard error before the signal **SIGKILL** is sent to the process. If a shared library cannot be mapped, the subroutine returns one of the following error codes:

Item	Description
ENOENT	One or more components of the path name of the shared library file do not exist.
ENOTDIR	A component of the path prefix of the shared library file is not a directory.

Item	Description
ENAMETOOLONG	A component of a path name prefix of a shared library file exceeded 255 characters, or an entire path name exceeded 1023 characters.
EACCES	Search permission is denied for a directory listed in the path prefix of the shared library file.
EACCES	The shared library file mode denies execution permission.
ENOEXEC	The shared library file has the appropriate access permission, but a magic number in its header is not valid.
ETXTBSY	The shared library file is currently open for writing by some other process.
ENOMEM	The shared library requires more memory than is allowed by the system-imposed maximum.
ESTALE	The process root or current directory is located in a virtual file system that has been unmounted.
EPROCLIM	If WLM is running, the limit on the number of processes, threads, or logins in the class may have been met.

If the **fexecve** subroutine is unsuccessful, it might also return one of the following error codes:

Item	Description
EBADF	The *FileDescriptor* argument does not specify a valid open file descriptor.
ENOENT	The *FileDescriptor* argument points to a shell procedure, but the original parent directory of the file has been deleted.

If NFS is installed on the system, the **exec** subroutine can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

exit, atexit, unatexit, _exit, or _Exit Subroutine

Purpose

Terminates a process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

void exit (*Status*)
int *Status*;

void _exit (*Status*)
int *Status*;

```
void _Exit (Status)
int Status;
```

```
#include <sys/limits.h>
```

int atexit (*Function*)
void (Function*) (void);**

```
int unatexit (Function)
void (*Function)(void);
```

Description

The **exit** subroutine terminates the calling process after calling the standard I/O library **_cleanup** function to flush any buffered output. Also, it calls any functions registered previously for the process by the **atexit** subroutine. The **atexit** subroutine registers functions called at normal process termination for cleanup processing. Normal termination occurs as a result of either a call to the **exit** subroutine or a **return** statement in the **main** function.

Each function a call to the **atexit** subroutine registers must return. This action ensures that all registered functions are called.

Finally, the **exit** subroutine calls the **_exit** subroutine, which completes process termination and does not return. The **_exit** subroutine terminates the calling process and causes the following to occur:

The **_Exit** subroutine is functionally equivalent to the **_exit** subroutine. The **_Exit** subroutine does not call functions registered with **atexit** or any registered signal handlers. The way the subroutine is implemented determines whether open streams are flushed or closed, and whether temporary files are removed. The calling process is terminated with the consequences described below.

- All of the file descriptors, directory streams, conversion descriptors, and message catalog descriptors open in the calling process are closed.
- If the parent process of the calling process is executing a **wait** or **waitpid**, and has not set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, it is notified of the calling process' termination and the low order eight bits (that is, bits 0377) of *status* are made available to it. If the parent is not waiting, the child's status is made available to it when the parent subsequently executes **wait** or **waitpid**.
- If the parent process of the calling process is not executing a **wait** or **waitpid**, and has neither set its SA_NOCLDWAIT flag nor set SIGCHLD to SIG_IGN, the calling process is transformed into a zombie process. A zombie process is an inactive process that is deleted at some later time when its parent process executes **wait** or **waitpid**.
- Termination of a process does not directly terminate its children. The sending of a SIGHUP signal indirectly terminates children in some circumstances. This can be accomplished in one of two ways. If the implementation supports the SIGCHLD signal, a SIGCHLD is sent to the parent process. If the parent process has set its SA_NOCLDWAIT flag, or set SIGCHLD to SIG_IGN, the status is discarded, and the lifetime of the calling process ends immediately. If SA_NOCLDWAIT is set, it is implementation defined whether a SIGCHLD signal is sent to the parent process.
- The parent process ID of all of the calling process' existing child processes and zombie processes are set to the process ID of an implementation defined system process.
- Each attached shared memory segment is detached and the value of *shm_nattch* (see **shmget**) in the data structure associated with its shared memory ID is decremented by 1.
- For each semaphore for which the calling process has set a *semadj* value (see **semop**), that value is added to the *semval* of the specified semaphore.
- If the process is a controlling process, the SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the calling process.
- If the process is a controlling process, the controlling terminal associated with the session is disassociated from the session, allowing it to be acquired by a new controlling process.
- If the exit of the process causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, a SIGHUP signal followed by a SIGCONT signal is sent to each process in the newly orphaned process group.

- All open named semaphores in the calling process are closed as if by appropriate calls to **sem_close**.
- Memory mappings that were created in the process are unmapped before the process is destroyed.
- Any blocks of typed memory that were mapped in the calling process are unmapped, as if the **munmap** subroutine was implicitly called to unmap them.
- All open message queue descriptors in the calling process are closed.
- Any outstanding cancelable asynchronous I/O operations may be canceled. Those asynchronous I/O operations that are not canceled complete as if the **_Exit** subroutine had not yet occurred, but any associated signal notifications are suppressed.

  The **_Exit** subroutine may block awaiting such I/O completion. The implementation defines whether any I/O is canceled, and which I/O may be canceled upon **_Exit**.
- Threads terminated by a call to **_Exit** do not invoke their cancelation cleanup handlers or per thread data destructors.
- If the calling process is a trace controller process, any trace streams that were created by the calling process are shut down.

The **unatexit** subroutine is used to unregister functions that are previously registered by the **atexit** subroutine. If the referenced function is found, it is removed from the list of functions that are called at normal program termination.

Parameters

Item	Description
Status	Indicates the status of the process. May be set to 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only the least significant 8 bits are available to a waiting parent process.
Function	Specifies a function to be called at normal process termination for cleanup processing. You may specify a number of functions to the limit set by the **ATEXIT_MAX** function, which is defined in the **sys/limits.h** file. A pushdown stack of functions is kept so that the last function registered is the first function called.

Return Values

Upon successful completion, the **atexit** subroutine returns a value of 0. Otherwise, a nonzero value is returned. The **exit** and **_exit** subroutines do not return a value.

The **unatexit()** subroutine returns a value of 0 if the function referenced by *Function* is found and removed from the **atexit** list. Otherwise, a nonzero value is returned.

exp, expf, expl, expd32, expd64, and expd128 Subroutines

Purpose

Computes exponential, logarithm, and power functions.

Libraries

IEEE Math Library (**libm.a**) or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
```

```
double exp ( x)
double x;
```

```
float expf (x)
float x;
```

```
long double expl (x)
long double x;
```

```
_Decimal32 expd32 (x)
_Decimal32 x;
_Decimal64 expd64 (x)
_Decimal64 x;

_Decimal128 expd128 (x)
_Decimal128 x;
```

Description

These subroutines are used to compute exponential, logarithm, and power functions.

The **exp**, **expf**, **expl**, **expd32**, **expd64**, and **expd128** subroutines returns `exp (x)`.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies some double-precision floating-point value.
y	Specifies some double-precision floating-point value.

Return Values

Upon successful completion, the **exp**, **expf**, **expl**, **expd32**, **expd64**, and **expd128** subroutines return the exponential value of *x*.

If the correct value would cause overflow, a range error occurs and the **exp**, **expf**, **expl**, **expd32**, **expd64**, and **expd128** subroutine returns the value of the macro **HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and either 0.0 (if supported), or an implementation-defined value is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0, 1 is returned.

If *x* is -Inf, +0 is returned.

If *x* is +Inf, *x* is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Error Codes

When using the **libm.a** library:

Item	Description
exp	If the correct value would overflow, the **exp** subroutine returns a **HUGE_VAL** value and the **errno** global variable is set to a **ERANGE** value.

When using **libmsaa.a**(**-lmsaa**):

Item	Description
exp	If the correct value would overflow, the **exp** subroutine returns a **HUGE_VAL** value. If the correct value would underflow, the **exp** subroutine returns 0. In both cases **errno** is set to **ERANGE**.
expl	If the correct value would overflow, the **expl** subroutine returns a **HUGE_VAL** value. If the correct value would underflow, the **expl** subroutine returns 0. In both cases **errno** is set to **ERANGE**.
expl	If the correct value overflows, the **expl** subroutine returns a **HUGE_VAL** value and **errno** is set to **ERANGE**.

These error-handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** library.

exp2, exp2f, exp2l, exp2d32, exp2d64, and exp2d128 Subroutines

Purpose

Computes the base 2 exponential.

Syntax

```
#include <math.h>

double exp2 (x)
double x;

float exp2f (x)
float x;

long double exp2l (x)
long double x;
_Decimal32 exp2d32 (x)
_Decimal32 x;

_Decimal64 exp2d64 (x)
_Decimal64 x;

_Decimal128 exp2d128 (x)
_Decimal128 x;
```

Description

The **exp2**, **exp2f**, **exp2l**, **exp2d32**, **exp2d64**, and **exp2d128** subroutines compute the base 2 exponential of the *x* parameter.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept (FE_ALL_EXCEPT)** before calling these subroutines. On return, if **errno** is nonzero or **fetestexcept (FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW)** is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the base 2 exponential to be computed.

Return Values

Upon successful completion, the **exp2**, **exp2f**, **exp2l**, **exp2d32**, **exp2d64**, or **exp2d128** subroutine returns 2^x .

If the correct value causes overflow, a range error occurs and the **exp2**, **exp2f**, **exp2l**, **exp2d32**, **exp2d64**, and **exp2d128** subroutines return the value of the macro (**HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively).

If the correct value causes underflow and is not representable, a range error occurs, and 0.0 is returned.

If *x* is NaN, NaN is returned.

If *x* is ±0, 1 is returned.

If *x* is -Inf, 0 is returned.

If *x* is +Inf, *x* is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

expm1, expm1f, expm1l, expm1d32, expm1d64, and expm1d128 Subroutine

Purpose

Computes exponential functions.

Syntax

```
#include <math.h>

float expm1f (x)
float x;

long double expm1l (x)
long double x;

double expm1 (x)
double x;
_Decimal32 expm1d32 (x)
_Decimal32 x;

_Decimal64 expm1d64 (x)
_Decimal64 x;
_Decimal128 expm1d128 (x)
_Decimal128 x;
```

Description

The **expm1f**, **expm1l**, **expm1**, **expm1d32**, **expm1d64**, and **expm1d128** subroutines compute e^x- 1.0.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **expm1f**, **expm1l**, **expm1**, **expm1d32**, **expm1d64**, and **expm1d128** subroutines return e^x- 1.0.

If the correct value would cause overflow, a range error occurs and the **expm1f**, **expm1l**, **expm1**, **expm1d32**, **expm1d64**, and **expm1d128** subroutines return the value of the macro **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±0, ±0 is returned.

If *x* is -Inf, -1 is returned.

If *x* is +Inf, *x* is returned.

If *x* is subnormal, a range error may occur and *x* is returned.

f

The following Base Operating System (BOS) runtime services begin with the letter *f*.

fabsf, fabsl, fabs, fabsd32, fabsd64, and fabsd128 Subroutines

Purpose

Determines the absolute value.

Syntax

```
#include <math.h>

float fabsf (x)
float x;

long double fabsl (x)
long double x;

double fabs (x)
double x;

_Decimal32 fabsd32 (x)
_Decimal32 x;

_Decimal64 fabsd64 (x)
_Decimal64 x;

_Decimal128 fabsd128 (x)
_Decimal128 x;
```

Description

The **fabsf**, **fabsl**, **fabs**, **fabsd32**, **fabsd64**, and **fabsd128** subroutines compute the absolute value of the *x* parameter, |*x*|.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **fabsf**, **fabsl**, **fabs**, **fabsd32**, **fabsd64**, and **fabsd128** subroutines return the absolute value of *x*.

If *x* is NaN, a NaN is returned.

If *x* is ±0, +0 is returned.

If *x* is ±Inf, +Inf is returned.

fattach Subroutine

Purpose

Attaches a STREAMS-based file descriptor to a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stropts.h>
int fattach(int fildes, const char *path);
```

Description

The **fattach** subroutine attaches a STREAMS-based file descriptor to a file, effectively associating a pathname with *fildes*. The *fildes* argument must be a valid open file descriptor associated with a STREAMS file. The *path* argument points to a pathname of an existing file. The process must have appropriate privileges, or must be the owner of the file named by *path* and have write permission. A successful call to **fattach** subroutine causes all pathnames that name the file named by *path* to name the STREAMS file associated with *fildes*, until the STEAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several pathnames associated with it.

The attributes of the named STREAMS file are initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If any attributes of the named STREAMS file are subsequently changed (for example, by **chmod** subroutine), neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes* refers are affected.

File descriptors referring to the underlying file, opened prior to an **fattach** subroutine, continue to refer to the underlying file.

Parameters

Item	Description
fildes	A file descriptor identifying an open STREAMS-based object.
path	An existing pathname which will be associated with *fildes*.

Return Value

Item	Description
0	Successful completion.
-1	Not successful and *errno* set to one of the following.

Errno Value

Item	Description
EACCES	Search permission is denied for a component of the path prefix, or the process is the owner of *path* but does not have write permission on the file named by *path*.
EBADF	The file referred to by *fildes* is not an open file descriptor.
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The effective user ID of the process is not the owner of the file named by *path* and the process does not have appropriate privilege.

Item	Description
EBUSY	The file named by *path* is currently a mount point or has a STREAMS file attached to it.
ENAMETOOLONG	The size of *path* exceeds {**PATH_MAX**}, or a component of *path* is longer than {**NAME_MAX**}.
ELOOP	Too many symbolic links wer encountered in resolving *path*.
EINVAL	The *fildes* argument does not refer to a STREAMS file.
ENOMEM	Insufficient storage space is available.

fchdir Subroutine

Purpose

Directory pointed to by the file descriptor becomes the current working directory.

Library

Standard C Library **(libc.a)**

Syntax

#include <unistd.h>

int fchdir (int *Fildes***)**

Description

The **fchdir** subroutine causes the directory specified by the *Fildes* parameter to become the current working directory.

Parameter

Item	Description
Fildes	A file descriptor identifying an open directory obtained from a call to the **open** subroutine.

Return Values

Item	Description
0	Successful completion
-1	Not successful and **errno** set to one of the following.

Error Codes

Item	Description
EACCES	Search access if denied.
EBADF	The file referred to by *Fildes* is not an open file descriptor.
ENOTDIR	The open file descriptor does not refer to a directory.

fclear or fclear64 Subroutine

Purpose

Makes a hole in a file.

Library

Standard C Library (**libc.a**)

Syntax

```
off_t fclear ( FileDescriptor,  NumberOfBytes)
int FileDescriptor;
off_t NumberOfBytes;

off64_t fclear64 ( FileDescriptor,  NumberOfBytes)
int FileDescriptor;
off64_t NumberOfBytes;
```

Description

The **fclear** and **fclear64** subroutines zero the number of bytes specified by the *NumberOfBytes* parameter starting at the current file pointer for the file specified in the *FileDescriptor* parameter. If Network File System (NFS) is installed on your system, this file can reside on another node.

The **fclear** subroutine can only clear up to **OFF_MAX** bytes of the file while **fclear64** can clear up to the maximum file size.

The **fclear** and **fclear64** subroutines cannot be applied to a file that a process has opened with the **O_DEFER** mode.

Successful completion of the **fclear** and **fclear64** subroutines clear the SetUserID bit (**S_ISUID**) of the file if any of the following are true:

- The calling process does not have root user authority.
- The effective user ID of the calling process does not match the user ID of the file.
- The file is executable by the group (**S_IXGRP**) or others (**S_IXOTH**).

This subroutine also clears the SetGroupID bit (**S_ISGID**) if:

- The file does not match the effective group ID or one of the supplementary group IDs of the process,

  OR
- The file is executable by the owner (**S_IXUSR**) or others (**S_IXOTH**).

  Note: Clearing of the SetUserID and SetGroupID bits can occur even if the subroutine fails because the data in the file was modified before the error was detected.

In the large file enabled programming environment, **fclear** is redefined to be **fclear64**.

Parameters

Item	Description
FileDescriptor	Indicates the file specified by the *FileDescriptor* parameter must be open for writing. The FileDescriptor is a small positive integer used instead of the file name to identify a file. This function differs from the logically equivalent write operation in that it returns full blocks of binary zeros to the file system, constructing holes in the file.

Item	Description
NumberOfBytes	Indicates the number of bytes that the seek pointer is advanced. If you use the **fclear** and **fclear64** subroutines past the end of a file, the rest of the file is cleared and the seek pointer is advanced by *NumberOfBytes*. The file size is updated to include this new hole, which leaves the current file position at the byte immediately beyond the new end-of-file pointer.

Return Values

Upon successful completion, a value of *NumberOfBytes* is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fclear** and **fclear64** subroutines fail if one or more of the following are true:

Item	Description
EIO	I/O error.
EBADF	The *FileDescriptor* value is not a valid file descriptor open for writing.
EINVAL	The file is not a regular file.
EMFILE	The file is mapped **O_DEFER** by one or more processes.
EAGAIN	The write operation in the **fclear** subroutine failed due to an enforced write lock on the file.

Item	Description
EFBIG	The current offset plus *NumberOfBytes* is exceeds the offset maximum established in the open file description associated with *FileDescriptor*.

Item	Description
EFBIG	An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. If the user has set the environment variable **XPG_SUS_ENV=ON** prior to execution of the process, then the **SIGXFSZ** signal is posted to the process when exceeding the process' file size limit.

If NFS is installed on the system the **fclear** and **fclear64** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

fclose or fflush Subroutine

Purpose

Closes or flushes a stream.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

int fclose (*Stream*)
FILE ***Stream*;

int fflush (*Stream*)
FILE ***Stream*;

Description

The **fclose** subroutine writes buffered data to the stream specified by the *Stream* parameter, and then closes the stream. The **fclose** subroutine is automatically called for all open files when the **exit** subroutine is invoked.

The **fflush** subroutine writes any buffered data for the stream specified by the *Stream* parameter and leaves the stream open. The **fflush** subroutine marks the `st_ctime` and `st_mtime` fields of the underlying file for update.

If the *Stream* parameter is a null pointer, the **fflush** subroutine performs this flushing action on all streams for which the behavior is defined.

Parameters

Item	Description
Stream	Specifies the output stream.

Return Values

Upon successful completion, the **fclose** and **fflush** subroutines return a value of 0. Otherwise, a value of EOF is returned.

Error Codes

If the **fclose** and **fflush** subroutines are unsuccessful, the following errors are returned through the **errno** global variable:

Item	Description
EAGAIN	The **O_NONBLOCK** or `O_NDELAY` flag is set for the file descriptor underlying the *Stream* parameter and the process would be delayed in the write operation.
EBADF	The file descriptor underlying *Stream* is not valid.
EFBIG	An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. See the **ulimit** subroutine.
EFBIG	The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
EINTR	The **fflush** subroutine was interrupted by a signal.
EIO	The process is a member of a background process group attempting to write to its controlling terminal, the **TOSTOP** signal is set, the process is neither ignoring nor blocking the **SIGTTOU** signal and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.
ENOMEM	The underlying stream was created by open_memstream() or open_wmemstream( ) and insufficient memory is available.

Item	Description
ENOSPC	No free space remained on the device containing the file or in the buffer used by the fmemopen( ) function.
EPIPE	An attempt is made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal is sent to the process.
ENXIO	A request was made of a non-existent device, or the request was outside the capabilities of the device

fcntl, dup, or dup2 Subroutine

Purpose

Controls open file descriptors.

Library

Standard C Library (**libc.a**)

Berkeley compatibility library (`libbsd.a`) (for the **fcntl** subroutine)

Syntax

#include <fcntl.h>

int fcntl (*FileDescriptor*, *Command*, *Argument*) **int** *FileDescriptor*, *Command*, *Argument*;

#include <unistd.h>

int dup2(*Old*, *New*) **int** *Old*, *New*;

int dup(*FileDescriptor*) **int** *FileDescriptor*;

Description

The **fcntl** subroutine performs controlling operations on the open file specified by the *FileDescriptor* parameter. If Network File System (NFS) is installed on your system, the open file can reside on another node. The **fcntl** subroutine is used to:

- Duplicate open file descriptors.
- Set and get the file-descriptor flags.
- Set and get the file-status flags.
- Manage record locks.
- Manage asynchronous I/O ownership.
- Close multiple files.

The **fcntl** subroutine can provide the same functions as the **dup** and **dup2** subroutines.

If *FileDescriptor* refers to a terminal device or socket, then asynchronous I/O facilities can be used. These facilities are normally enabled by using the **ioctl** subroutine with the **FIOASYNC**, **FIOSETOWN**, and **FIOGETOWN** commands. However, a BSD-compatible mechanism is also available if the application is linked with the **libbsd.a** library.

When the *FileDescriptor* parameter refers to a shared memory object, the **fcntl** subroutine manages only the **F_DUPFD**, **F_DUP2FD**, **F_GETFD**, **F_SETFD**, **F_GETFL**, and **F_CLOSEM** commands.

When using the **libbsd.a** library, asynchronous I/O is enabled by using the **F_SETFL** command with the **FASYNC** flag set in the *Argument* parameter. The **F_GETOWN** and **F_SETOWN** commands get the current

asynchronous I/O owner and set the asynchronous I/O owner. However, these commands are valid only when the file descriptor refers to a terminal device or a socket.

All applications containing the **fcntl** subroutine must be complied with **_BSD** set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

General Record Locking Information

A lock is either an *enforced* or *advisory lock* and either a *read* or a *write lock*.

> **Attention:** Buffered I/O does not work properly when used with file locking. Do not use the standard I/O package routines on files that are going to be locked.

For a lock to be an enforced lock, the Enforced Locking attribute of the file must be set; for example, the **S_ENFMT** bit must be set, but the **S_IXGRP**, **S_IXUSR**, and **S_IXOTH** bits must be clear. Otherwise, the lock is an advisory lock. A given file can have advisory or enforced locks, but not both. The description of the **sys/mode.h** file includes a description of file attributes.

When a process holds an enforced lock on a section of a file, no other process can access that section of the file with the **read** or **write** subroutine. In addition, the **open** and **ftruncate** subroutines cannot truncate the locked section of the file, and the **fclear** subroutine cannot modify the locked section of the file. If another process attempts to read or modify the locked section of the file, the process either sleeps until the section is unlocked or returns with an error indication.

When a process holds an advisory lock on a section of a file, no other process can lock that section of the file (or an overlapping section) with the **fcntl** subroutine. (No other subroutines are affected.) As a result, processes must voluntarily call the **fcntl** subroutine in order to make advisory locks effective.

When a process holds a read lock on a section of a file, other processes can also set read locks on that section or on subsets of it. Read locks are also called *shared* locks.

A read lock prevents any other process from setting a write lock on any part of the protected area. If the read lock is also an enforced lock, no other process can modify the protected area.

The file descriptor on which a read lock is being placed must have been opened with read access.

When a process holds a write lock on a section of a file, no other process can set a read lock or a write lock on that section. Write locks are also called *exclusive* locks. Only one write lock and no read locks can exist for a specific section of a file at any time.

If the lock is also an enforced lock, no other process can read or modify the protected area.

The following general rules about file locking apply:

- Changing or unlocking part of a file in the middle of a locked section leaves two smaller sections locked at each end of the originally locked section.
- If the calling process holds a lock on a file, that lock can be replaced by later calls to the **fcntl** subroutine.
- All locks associated with a file for a given process are removed when the process closes *any* file descriptor for that file.
- Locks are not inherited by a child process after a **fork** subroutine is run.

  Note: Deadlocks due to file locks in a distributed system are not always detected. When such deadlocks can possibly occur, the programs requesting the locks should set time-out timers.

Locks can start and extend beyond the current end of a file but cannot be negative relative to the beginning of the file. A lock can be set to extend to the end of the file by setting the `l_len` field to 0. If such a lock also has the `l_start` and `l_whence` fields set to 0, the whole file is locked. The `l_len`, `l_start`, and `l_whence` locking fields are part of the **flock** structure.

When an application locks a region of a file using the 32 bit locking interface (F_SETLK), and the last byte of the lock range includes MAX_OFF (2 Gb - 1), then the lock range for the unlock request will be extended to include MAX_END (2 ^ ^ 63 - 1).

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor obtained from a successful call to the **open** subroutine, **fcntl** subroutine, **pipe** subroutine, or **shm_open** subroutine. File descriptors are small positive integers used (instead offile names) to identify files or a shared memory object.
Argument	Specifies a variable whose value sets the function specified by the *Command* parameter. When dealing with file locks, the *Argument* parameter must be a pointer to the **FLOCK** structure.
Command	Specifies the operation performed by the **fcntl** subroutine. The **fcntl** subroutine can duplicate open file descriptors, set file-descriptor flags, set file descriptor locks, set process IDs, and close open file descriptors.

Duplicating File Descriptors

Item	Description
F_DUPFD	Returns a new file descriptor as follows: • Lowest-numbered available file descriptor greater than or equal to the *Argument* parameter • Same object references as the original file • Same file pointer as the original file (that is, both file descriptors share one file pointer if the object is a file) • Same access mode (read, write, or read-write) • Same file status flags (That is, both file descriptors share the same file status flags.) • The **close-on-exec** flag (**FD_CLOEXEC** bit) associated with the new file descriptor is cleared

Setting File-Descriptor Flags

Item	Description
F_GETFD	Gets the **close-on-exec** flag (**FD_CLOEXEC** bit) that is associated with the file descriptor specified by the *FileDescriptor* parameter. The *Argument* parameter is ignored. File descriptor flags are associated with a single file descriptor, and do not affect others associated with the same file.
F_SETFD	Assigns the value of the *Argument* parameter to the **close-on-exec** flag (**FD_CLOEXEC** bit) that is associated with the *FileDescriptor* parameter. If the **FD_CLOEXEC** flag value is 0, the file remains open across any calls to **exec** subroutines; otherwise, the file will close upon the successful execution of an **exec** subroutine.

Item	Description
F_GETFL	Gets the file-status flags and file-access modes for the open file description associated with the file descriptor specified by the *FileDescriptor* parameter. The open file description is set at the time the file is opened and applies only to those file descriptors associated with that particular call to the file. This open file descriptor does not affect other file descriptors that refer to the same file with different open file descriptions. The file-status flags have the following values: **O_APPEND** Set append mode. **O_NONBLOCK** No delay. The file-access modes have the following values: **O_RDONLY** Open for reading only. **O_RDWR** Open for reading and writing. **O_WRONLY** Open for writing only. The file access flags can be extracted from the return value using the **O_ACCMODE** mask, which is defined in the **fcntl.h** file.
F_SETFL	Sets the file status flags from the corresponding bits specified by the *Argument* parameter. The file-status flags are set for the open file description associated with the file descriptor specified by the *FileDescriptor* parameter. The following flags may be set: • **O_APPEND** or **FAPPEND** • **O_NDELAY** or **FNDELAY** • **O_NONBLOCK** or **FNONBLOCK** • **O_SYNC** or **FSYNC** • **FASYNC** The **O_NDELAY** and **O_NONBLOCK** flags affect only operations against file descriptors derived from the same **open** subroutine. In BSD, these operations apply to all file descriptors that refer to the object.

Setting File Locks

Item	Description
F_GETLK	Gets information on the first lock that blocks the lock described in the **flock** structure. The *Argument* parameter should be a pointer to a type **struct flock**, as defined in the **flock.h** file. The information retrieved by the **fcntl** subroutine overwrites the information in the **struct flock** pointed to by the *Argument* parameter. If no lock is found that would prevent this lock from being created, the structure is left unchanged, except for lock type (`l_type`) which is set to **F_UNLCK**.
F_SETLK	Sets or clears a file-segment lock according to the lock description pointed to by the *Argument* parameter. The *Argument* parameter should be a pointer to a type **struct flock**, which is defined in the **flock.h** file. The **F_SETLK** option is used to establish read (or shared) locks (**F_RDLCK**), or write (or exclusive) locks (**F_WRLCK**), as well as to remove either type of lock (**F_UNLCK**). The lock types are defined by the **fcntl.h** file. If a shared or exclusive lock cannot be set, the **fcntl** subroutine returns immediately.

Item	Description
F_SETLKW	Performs the same function as the **F_SETLK** option unless a read or write lock is blocked by existing locks, in which case the process sleeps until the section of the file is free to be locked. If a signal that is to be caught is received while the **fcntl** subroutine is waiting for a region, the **fcntl** subroutine is interrupted, returns a -1, sets the **errno** global variable to **EINTR**. The lock operation is not done.

Item	Description
F_GETLK64	Gets information on the first lock that blocks the lock described in the **flock64** structure. The *Argument* parameter should be a pointer to an object of the type **struct flock64**, as defined in the **flock.h** file. The information retrieved by the **fcntl** subroutine overwrites the information in the **struct flock64** pointed to by the *Argument* parameter. If no lock is found that would prevent this lock from being created, the structure is left unchanged, except for lock type (`l_type`) which is set to **F_UNLCK**.
F_SETLK64	Sets or clears a file-segment lock according to the lock description pointed to by the *Argument* parameter. The *Argument* parameter should be a pointer to a type **struct flock64**, which is defined in the **flock.h** file. The **F_SETLK** option is used to establish read (or shared) locks (**F_RDLCK**), or write (or exclusive) locks (**F_WRLCK**), as well as to remove either type of lock (**F_UNLCK**). The lock types are defined by the **fcntl.h** file. If a shared or exclusive lock cannot be set, the **fcntl** subroutine returns immediately.
F_SETLKW64	Performs the same function as the **F_SETLK** option unless a read or write lock is blocked by existing locks, in which case the process sleeps until the section of the file is free to be locked. If a signal that is to be caught is received while the **fcntl** subroutine is waiting for a region, the **fcntl** subroutine is interrupted, returns a -1, sets the **errno** global variable to **EINTR**. The lock operation is not done.

Setting Process ID

Item	Description
F_GETOWN	Gets the process ID or process group currently receiving **SIGIO** and **SIGURG** signals. Process groups are returned as negative values.
F_SETOWN	Sets the process or process group to receive **SIGIO** and **SIGURG** signals. Process groups are specified by supplying a negative *Argument* value. Otherwise, the *Argument* parameter is interpreted as a process ID.

Closing File Descriptors

Item	Description
F_CLOSEM	Closes all file descriptors from *FileDescriptor* up to the highest currently open file descriptor (*U_maxofile*).
Old	Specifies an open file descriptor.
New	Specifies an open file descriptor that is returned by the **dup2** subroutine.

Compatibility Interfaces

The lockfx Subroutine

The **fcntl** subroutine functions similar to the **lockfx** subroutine, when the *Command* parameter is **F_SETLK**, **F_SETLKW**, or **F_GETLK**, and when used in the following way:

fcntl (*FileDescriptor***,** *Command***,** *Argument***)**

is equivalent to:

lockfx (*FileDescriptor***,** *Command***,** *Argument***)**

The dup and dup2 Subroutines

The **fcntl** subroutine functions similar to the **dup** and **dup2** subroutines, when used in the following way:

```
dup (FileDescriptor)
```

is equivalent to:

```
fcntl (FileDescriptor, F_DUPFD, 0)
```

```
dup2 (Old, New)
```

is equivalent to:

```
close (New);
fcntl(Old, F_DUPFD, New)
```

The **dup** and **dup2** subroutines differ from the **fcntl** subroutine in the following ways:

- If the file descriptor specified by the *New* parameter is greater than or equal to **OPEN_MAX**, the **dup2** subroutine returns a -1 and sets the **errno** variable to **EBADF**.
- If the file descriptor specified by the *Old* parameter is valid and equal to the file descriptor specified by the *New* parameter, the **dup2** subroutine will return the file descriptor specified by the *New* parameter, without closing it.
- If the file descriptor specified by the *Old* parameter is not valid, the **dup2** subroutine will be unsuccessful and will not close the file descriptor specified by the *New* parameter.
- The value returned by the **dup** and **dup2** subroutines is equal to the *New* parameter upon successful completion; otherwise, the return value is -1.

Return Values

Upon successful completion, the value returned depends on the value of the *Command* parameter, as follows:

Item	Description
Command	Return Value
F_DUPFD	A new file descriptor
F_GETFD	The value of the flag (only the **FD_CLOEXEC** bit is defined)
F_SETFD	A value other than -1
F_GETFL	The value of file flags
F_SETFL	A value other than -1
F_GETOWN	The value of descriptor owner
F_SETOWN	A value other than -1
F_GETLK	A value other than -1
F_SETLK	A value other than -1
F_SETLKW	A value other than -1
F_CLOSEM	A value other than -1.

If the **fcntl** subroutine fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fcntl** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EACCES	The *Command* argument is **F_SETLK**; the type of lock is a shared or exclusive lock and the segment of a file to be locked is already exclusively-locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.
EBADF	The *FileDescriptor* parameter is not a valid open file descriptor.
EDEADLK	The *Command* argument is **F_SETLKW**; the lock is blocked by some lock from another process and putting the calling process to sleep, waiting for that lock to become free would cause a deadlock.
ENOTTY	The file descriptor does not refer to a terminal device or socket.
EMFILE	The *Command* parameter is **F_DUPFD**, and the maximum number of file descriptors are currently open (**OPEN_MAX**).
EINVAL	The *Command* parameter is **F_DUPFD**, and the *Argument* parameter is negative or greater than or equal to **OPEN_MAX**.
EINVAL	An illegal value was provided for the *Command* parameter.
EINVAL	An attempt was made to lock a fifo or pipe.
ESRCH	The value of the *Command* parameter is **F_SETOWN**, and the process ID specified as the *Argument* parameter is not in use.
EINTR	The *Command* parameter was **F_SETLKW** and the process received a signal while waiting to acquire the lock.
EOVERFLOW	The *Command* parameter was **F_GETLK** and the block lock could not be represented in the **flock** structure.

The **dup** and **dup2** subroutines fail if one or both of the following are true:

Item	Description
EBADF	The *Old* parameter specifies an invalid open file descriptor or the *New* parameter specifies a file descriptor that is out of range.
EMFILE	The number of file descriptors exceeds the **OPEN_MAX** value or there is no file descriptor above the value of the *New* parameter.

If NFS is installed on the system, the **fcntl** subroutine can fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

fdetach Subroutine

Purpose

Detaches STREAMS-based file from the file to which it was attached.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stropts.h>
int fdetach(const char *path);
```

Parameters

Item	Description
path	Pathname of a file previous associated with a STREAMS-based object using the **fattach** subroutine.

Description

The **fdetach** subroutine detaches a STREAMS-based file from the file to which it was attached by a previous call to **fattach** subroutine. The *path* argument points to the pathname of the attached STREAMS file. The process must have appropriate privileges or be the owner of the file. A successful call to **fdetach** subroutine causes all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on *path* will operate on the underlying file and not on the STREAMS file.

All open file descriptors established while the STREAMS file was attached to the file referenced by *path* will still refer to the STREAMS file after the **fdetach** subroutine has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to **fdetach** subroutine has the same effect as performing the last **close** subroutine on the attached file.

The **umount** command may be used to detach a file name if an | application exits before performing **fdetach** subroutine.

Return Value

Item	Description
0	Successful completion.
-1	Not successful and **errno** set to one of the following.

Errno Value

Item	Description
EACCES	Search permission is denied on a component of the path prefix.
EPERM	The effective user ID is not the owner of *path* and the process does not have appropriate privileges.
ENOTDIR	A component of the path prefix is not a directory.
ENOENT	A component of *path* parameter does not name an existing file or *path* is an empty string.
EINVAL	The *path* parameter names a file that is not currently attached.
ENAMETOOLONG	The size of *path* parameter exceeds {**PATH_MAX**}, or a component of *path* is longer than {**NAME_MAX**}.
ELOOP	Too many symbolic links were encountered in resolving the *path* parameter.
ENOMEM	Insufficient storage space is available.

fdim, fdimf, fdiml, fdimd32, fdimd64, and fdimd128 Subroutines

Purpose

Computes the positive difference between two floating-point numbers.

Syntax

```
#include <math.h>

double fdim (x, y)
double x;
double y;

float fdimf (x, y)
float x;
float y;

long double fdiml (x, y)
long double x;
long double y;

_Decimal32 fdimd32 (x, y);
_Decimal32 x;
_Decimal32 y;

_Decimal64 fdimd64 (x, y);
_Decimal64 x;
_Decimal64 y;

_Decimal128 fdimd128 (x, y);
_Decimal128 x;
_Decimal128 y;
```

Description

The **fdim**, **fdimf**, **fdiml**, **fdimd32**, **fdimd64**, and **fdimd128** subroutines determine the positive difference between their arguments. If the value of the *x* parameter is greater than that of the *y* parameter, *x* - *y* is returned. If *x* is less than or equal to *y*, +0 is returned.

An application that wants to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. On return, if the **errno** is a value of non-zero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is a value of non-zero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.
y	Specifies the value to be computed.

Return Values

Upon successful completion, the **fdim**, **fdimf**, **fdiml**, **fdimd32**, **fdimd64**, and **fdimd128** subroutines return the positive difference value.

If *x*-*y* is positive and overflows, a range error occurs and the **fdim**, **fdimf**, **fdiml**, **fdimd32**, **fdimd64**, and **fdimd128** subroutines return the value of the **HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64** and **HUGE_VAL_D128** macro respectively.

If *x*-*y* is positive and underflows, a range error might occur, and 0.0 is returned.

If *x* or *y* is NaN, a NaN is returned.

fe_dec_getround and fe_dec_setround Subroutines

Purpose

Reads and sets the IEEE decimal floating-point rounding mode.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fenv.h>
int fe_dec_getround ();
int fe_dec_setround (RoundMode);
int RoundMode
```

Description

The **fe_dec_getround** subroutine returns the current rounding mode. The **fe_dec_setround** subroutine changes the rounding mode to the *RoundMode* parameter and returns the value of zero if it successfully completes.

Decimal floating-point rounding occurs when the infinitely precise result of a decimal floating-point operation cannot be represented exactly in the destination decimal floating-point format. The IEEE Standard for decimal floating-point arithmetic defines five modes that round the floating-point numbers: **round toward zero**, **round to nearest**, **round toward +INF**, **round toward -INF**, and **round to nearest ties away from zero**. Once a rounding mode is selected, it affects all subsequent decimal floating-point operations until another rounding mode is selected.

Tip: The default decimal floating-point rounding mode is the **round to nearest** mode. All C main programs begin with the rounding mode that is set to **round to nearest**.

The encodings of the rounding modes are defined in the ANSI C Standard. The **fenv.h** file contains definitions for the rounding modes. The following table shows the **fenv.h** definition, the ANSI C Standard value, and a description of each rounding mode.

fenv.h definition	ANSI value	Description
FE_DEC_TONEAREST	0	Round to nearest
FE_DEC_TOWARDZERO	1	Round toward zero
FE_DEC_UPWARD	2	Round toward +INF
FE_DEC_DOWNWARD	3	Round toward -INF
FE_DEC_TONEARESTFROMZERO	4	Round to nearest ties away from zero

Parameters

Item	Description
RoundMode	Specifies one of the following modes: FE_DEC_TOWARDZERO, FE_DEC_TONEAREST, FE_DEC_UPWARD, FE_DEC_DOWNWARD, FE_DEC_TONEARESTFROMZERO.

Return Values

On successful completion, the **fe_dec_getround** subroutine returns the current rounding mode. Otherwise , it returns -1.

On successful completion, the **fe_dec_setround** subroutine returns the value of zero. Otherwise, it returns -1.

feclearexcept Subroutine

Purpose

Clears floating-point exceptions.

Syntax

```
#include <fenv.h>

int feclearexcept (excepts)
int excepts;
```

Description

The **feclearexcept** subroutine attempts to clear the supported floating-point exceptions represented by the *excepts* parameter.

Parameters

Item	Description
excepts	Specifies the supported floating-point exception to be cleared.

Return Values

If the *excepts* parameter is zero or if all the specified exceptions were successfully cleared, the **feclearexcept** subroutine returns zero. Otherwise, it returns a nonzero value.

fegetenv or fesetenv Subroutine

Purpose

Gets and sets the current floating-point environment.

Syntax

```
#include <fenv.h>

int fegetenv (envp)
fenv_t *envp;

int fesetenv (envp)
const fenv_t *envp;
```

Description

The **fegetenv** subroutine stores the current floating-point environment in the object pointed to by the *envp* parameter.

The **fesetenv** subroutine attempts to establish the floating-point environment represented by the object pointed to by the *envp* parameter. The *envp* parameter points to an object set by a call to the **fegetenv** or **feholdexcept** subroutines, or equal a floating-point environment macro. The **fesetenv** subroutine does not raise floating-point exceptions. It only installs the state of the floating-point status flags represented through its argument.

Parameters

Item	Description
envp	Points to an object set by a call to the **fegetenv** or **feholdexcept** subroutines, or equal a floating-point environment macro.

Return Values

If the representation was successfully stored, the **fegetenv** subroutine returns zero. Otherwise, it returns a nonzero value. If the environment was successfully established, the **fesetenv** subroutine returns zero. Otherwise, it returns a nonzero value.

fegetexceptflag or fesetexceptflag Subroutine

Purpose

Gets and sets floating-point status flags.

Syntax

```
#include <fenv.h>

int fegetexceptflag (flagp, excepts)
feexcept_t *flagp;
int excepts;

int fesetexceptflag (flagp, excepts)
const fexcept_t *flagp;
int excepts;
```

Description

The **fegetexceptflag** subroutine attempts to store an implementation-defined representation of the states of the floating-point status flags indicated by the *excepts* parameter in the object pointed to by the *flagp* parameter.

The **fesetexceptflag** subroutine attempts to set the floating-point status flags indicated by the *excepts* parameter to the states stored in the object pointed to by the *flagp* parameter. The value pointed to by the *flagp* parameter shall have been set by a previous call to the **fegetexceptflag** subroutine whose second argument represented at least those floating-point exceptions represented by the *excepts* parameter. This subroutine does not raise floating-point exceptions. It only sets the state of the flags.

Parameters

Item	Description
flagp	Points to the object that holds the implementation-defined representation of the states of the floating-point status flags.
excepts	Points to an implementation-defined representation of the states of the floating-point status flags.

Return Values

If the representation was successfully stored, the **fegetexceptflag** parameter returns zero. Otherwise, it returns a nonzero value. If the *excepts* parameter is zero or if all the specified exceptions were successfully set, the **fesetexceptflag** subroutine returns zero. Otherwise, it returns a nonzero value.

fegetround or fesetround Subroutine

Purpose

Gets and sets the current rounding direction.

Syntax

```
#include <fenv.h>

int fegetround (void)

int fesetround (round)
int round;
```

Description

The **fegetround** subroutine gets the current rounding direction.

The **fesetround** subroutine establishes the rounding direction represented by the *round* parameter. If the *round* parameter is not equal to the value of a rounding direction macro, the rounding direction is not changed.

Parameters

Item	Description
round	Specifies the rounding direction.

Return Values

The **fegetround** subroutine returns the value of the rounding direction macro representing the current rounding direction or a negative value if there is no such rounding direction macro or the current rounding direction is not determinable.

The **fesetround** subroutine returns a zero value if the requested rounding direction was established.

feholdexcept Subroutine

The **feholdexcept** subroutine returns zero if non-stop floating-point exception handling was successfully installed.

Purpose

Saves current floating-point environment.

Syntax

```
#include <fenv.h>

int feholdexcept (envp)
fenv_t *envp;
```

Description

The **feholdexcept** subroutine saves the current floating-point environment in the object pointed to by *envp*, clears the floating-point status flags, and installs a non-stop (continue on floating-point exceptions) mode for all floating-point exceptions.

Parameters

Item	Description
envp	Points to the current floating-point environment.

Return Values

fence Subroutine

Purpose

Allows you to request and change the virtual shared disk fence map.

Syntax

```
#include <vsd_ioctl.h>
int ioctl(FileDescriptor, Command, Argument)
int FileDescriptor, Command;
void *Argument;
```

Description

Use this subroutine to request and change the virtual shared disk fence map. The fence map, which controls whether virtual shared disks can send or satisfy requests from virtual shared disks at remote nodes, is defined as:

```
struct  vsd_FenceMap             /* This is the argument to the VSD fence ioctl. */
{
   ulong                  flags;
   vsd_minorBitmap_t      minornoBitmap; /* Bitmap of minor numbers to fence
                                            (supports 10000 vsds)              */
   vsd_Fence_Bitmap_t     nodesBitmap;   /* Nodes to (un)fence these vsds from
                                            (supports node numbers 1-2048)     */

}vsd_FenceMap_t
```

The flags **VSD_FENCE** and **VSD_UNFENCE** are mutually exclusive — an ioctl can either fence a set of virtual shared disks or unfence a set of virtual shared disks, but not both. The *minornoBitmap* denotes which virtual shared disks are to be fenced/unfenced from the nodes specified in the *nodesBitmap*.

Parameters

FileDescriptor
Specifies the open file descriptor for which the control operation is to be performed.

Command
Specifies the control function to be performed. The value of this parameter is always `GIOCFENCE`.

Argument
Specifies a pointer to a `vsd_fence_map` structure.

The *flags* field of the `vsd_fence_map` structure determines the type of operation that is performed. The flags could be set with one or more options using the OR operator. These options are as follows:

VSD_FENCE_FORCE
If this option is specified, a node can unfence itself.

VSD_FENCE_GET
Denotes a query request.

VSD_FENCE
Denotes a fence request.

VSD_UNFENCE
Denotes an unfence request.

Examples

The following example fences a virtual shared disk with a minor number of 7 from node 4 and 5, and unfences a virtual shared disk with a minor number of 5 from node 1:

```
int fd;
vsd_FenceMap_t FenceMap;

/* Clear the FenceMap */
bzero(FenceMap, sizeof(vsd_FenceMap_t));

/* fence nodes 4,5 from minor 7 */
FenceMap.flags = VSD_FENCE;
MAP_SET(7, FenceMap.minornoBitmap);
MAP_SET(4, FenceMap.nodesBitmap);
MAP_SET(5, FenceMap.nodesBitmap);

/* Issue the fence request */
ioctl(fd,GIOCFENCE,&FenceMap);

/* Unfence node 1 from minor 5*/
bzero(FenceMap, sizeof(vsd_FenceMap_t));
FenceMap.flags = VSD_UNFENCE | VSD_FENCE_FORCE;
MAP_SET(5, FenceMap.minornoBitmap);
MAP_SET(1, FenceMap.nodesBitmap);

/* Issue the fence request */
ioctl(fd,GIOCFENCE,&FenceMap);
```

Return Values

If the request succeeds, the ioctl returns 0. In the case of an error, a value of -1 is returned with the global variable `errno` set to identify the error.

Error Values

The `fence` ioctl subroutine can return the following error codes:

EACCES
Indicates that an unfence was requested from a fenced node without the VSD_FENCE_FORCE option.

EINVAL
Indicates an invalid request (ambiguous flags or unidentified virtual shared disks).

ENOCONNECT
Indicates that either the primary or the secondary node for a virtual shared disk to be fenced is not a member of the virtual shared disk group, or the virtual shared disk in question is in the **stopped** state.

ENOTREADY
Indicates that the group is not active or the Recoverable virtual shared disk subsystem is not available.

ENXIO
Indicates that the Virtual shared disk driver is being unloaded.

feof, ferror, clearerr, or fileno Macro

Purpose

Checks the status of a stream.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

int feof (*Stream*)
**FILE ****Stream*;

```
int ferror (Stream)
FILE *Stream;
```

```
void clearerr (Stream)
FILE *Stream;
```

```
int fileno (Stream)
FILE *Stream;
```

Description

The **feof** macro inquires about the end-of-file character (EOF). If EOF has previously been detected reading the input stream specified by the *Stream* parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **ferror** macro inquires about input or output errors. If an I/O error has previously occurred when reading from or writing to the stream specified by the *Stream* parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **clearerr** macro inquires about the status of a stream. The **clearerr** macro resets the error indicator and the EOF indicator to a value of 0 for the stream specified by the *Stream* parameter.

The **fileno** macro inquires about the status of a stream. The **fileno** macro returns the integer file descriptor associated with the stream pointed to by the *Stream* parameter. Otherwise a value of -1 is returned.

Parameters

Item	Description
Stream	Specifies the input or output stream.

feraiseexcept Subroutine

If the argument is zero or if all the specified exceptions were successfully raised, the **feraiseexcept** subroutine returns a zero. Otherwise, it returns a nonzero value.

Purpose

Raises the floating-point exception.

Syntax

```
#include <fenv.h>

int feraiseexcept (excepts)
int excepts;
```

Description

The **feraiseexcept** subroutine attempts to raise the supported floating-point exceptions represented by the *excepts* parameter. The order in which these floating-point exceptions are raised is unspecified.

Parameters

Item	Description
excepts	Points to the floating-point exceptions.

Return Values

fetch_and_add and fetch_and_addlp Subroutines

Purpose

Updates a variable atomically.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/atomic_op.h>
int fetch_and_add ( addr,  value)
atomic_p addr;
int value;

long fetch_and_addlp ( addr,  value)
atomic_l addr;
ulong value;
```

Description

The **fetch_and_add** and **fetch_and_addlp** subroutines increment one word in a single atomic operation. This operation is useful when a counter variable is shared between several threads or processes. When updating such a counter variable, it is important to make sure that the fetch, update, and store operations occur atomically (are not interruptible). For example, consider the sequence of events which could occur if the operations were interruptible:

1. A process fetches the counter value and adds one to it.
2. A second process fetches the counter value, adds one, and stores it.
3. The first process stores its value.

The result of this is that the update made by the second process is lost.

Traditionally, atomic access to a shared variable would be controlled by a mechanism such as semaphores. Compared to such mechanisms, the **fetch_and_add** and **fetch_and_addlp** subroutines require very little increase in processor usage.

For 32-bit applications, the **fetch_and_add** and **fetch_and_addlp** subroutines are identical and operate on a word aligned single word (32-bit variable aligned on a 4-byte boundary).

For 64-bit applications, the **fetch_and_add** subroutine operates on a word aligned single word (32-bit variable aligned on a 4-byte boundary) and the **fetch_and_addlp** subroutine operates on a double word aligned double word (64-bit variable aligned on an 8-byte boundary).

Parameters

Item	Description
addr	Specifies the address of the variable to be incremented.
value	Specifies the value to be added to the variable.

Return Values

This subroutine returns the original value of the variable.

fetch_and_and, fetch_and_or, fetch_and_andlp, and fetch_and_orlp Subroutines

Purpose

Sets or clears bits in a variable atomically.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/atomic_op.h>
uint fetch_and_and ( addr, mask)
atomic_p addr;
unit mask;

ulong fetch_and_andlp ( addr, mask)
atomic_l addr;
ulong mask;

uint fetch_and_or ( addr,mask)
atomic_p addr;
intunit mask;

ulong fetch_and_orlp ( addr, mask)
atomic_l addr;
ulong mask;
```

Description

The **fetch_and_and**, **fetch_and_andlp**, **fetch_and_or**, and **fetch_and_orlp** subroutines respectively clear and set bits in a variable, according to a bit *mask*, as a single atomic operation.

The **fetch_and_and** and **fetch_and_andlp** subroutines clear bits in the variable that correspond to clear bits in the bit mask.

The **fetch_and_or** and **fetch_and_orlp** subroutines sets bits in the variable that correspond to set bits in the bit mask.

For 32-bit applications, the **fetch_and_and** and **fetch_and_andlp** subroutines are identical and operate on a word aligned single word (32-bit variable aligned on a 4-byte boundary). The **fetch_and_or** and

fetch_and_orlp subroutines are identical and operate on a word aligned single word (32-bit variable aligned on a 4-byte boundary).

For 64-bit applications, the **fetch_and_and** and **fetch_and_or** operate on a word aligned single word (32-bit variable aligned on a 4-byte boundary). The **fetch_and_addlp** and **fetch_and_orlp** subroutines operate on a double word aligned double word (64-bit variable aligned on an 8 -byte boundary).

These operations are useful when a variable containing bit flags is shared between several threads or processes. When updating such a variable, it is important that the fetch, bit clear or set, and store operations occur atomically (are not interruptible). For example, consider the sequence of events which could occur if the operations were interruptible:

1. A process fetches the flags variable and sets a bit in it.
2. A second process fetches the flags variable, sets a different bit, and stores it.
3. The first process stores its value.

The result is that the update made by the second process is lost.

Traditionally, atomic access to a shared variable would be controlled by a mechanism such as semaphores. Compared to such mechanisms, the **fetch_and_and**, **fetch_and_andlp**, **fetch_and_or**, and **fetch_and_orlp** subroutines require very little overhead.

Parameters

Item	Description
addr	Specifies the address of the variable whose bits are to be cleared or set.
mask	Specifies the bit mask which is to be applied to the variable.

Return Values

These subroutines return the original value of the variable.

fetestexcept Subroutine

The **fetestexcept** subroutine determines which of a specified subset of the floating-point exception flags are currently set. The *excepts* parameter specifies the floating-point status flags to be queried.

The **fetestexcept** subroutine returns the value of the bitwise-inclusive OR of the floating-point exception macros corresponding to the currently set floating-point exceptions included in *excepts*.

Purpose

Tests floating-point exception flags.

Syntax

```
#include <fenv.h>

int fetestexcept (excepts)
int excepts;
```

Description

Parameters

Item	Description
excepts	Specifies the floating-point status flags to be queried.

Return Values

feupdateenv Subroutine

Purpose

Updates floating-point environment.

Syntax

```
#include <fenv.h>

int feupdateenv (envp)
const fenv_t *envp;
```

Description

The **feupdateenv** subroutine attempts to save the currently raised floating-point exceptions in its automatic storage, attempts to install the floating-point environment represented by the object pointed to by the *envp* parameter, and attempts to raise the saved floating-point exceptions. The *envp* parameter point to an object set by a call to **feholdexcept** or **fegetenv**, or equal a floating-point environment macro.

Parameters

Item	Description
envp	Points to an object set by a call to the **feholdexcept** or the **fegetenv** subroutine, or equal a floating-point environment macro.

Return Values

The **feupdateenv** subroutine returns a zero value if all the required actions were successfully carried out.

finfo or ffinfo Subroutine

Purpose

Returns file information.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/finfo.h>
```

```
int finfo(Path1, cmd, buffer, length)
const char *Path1;
int cmd;
void  *buffer;
int length;
```

```
int ffinfo (fd, cmd, buffer, length)
int fd;
int cmd;
void  *buffer;
int length;
```

Description

The **finfo** and **ffinfo** subroutines return specific file information for the specified file.

Parameters

Item	Description
Path1	Path name of a file system object to query.
fd	File descriptor for an open file to query.
cmd	Specifies the type of file information to be returned.
buffer	User supplied buffer which contains the file information upon successful return. **/usr/include/sys/finfo.h** describes the buffer.
length	Length of the query buffer.

Commands

Item	Description
FI_PATHCONF	When the **FI_PATHCONF** command is specified, a file's implementation information is returned. **Note:** The operating system provides another subroutine that retrieves file implementation characteristics, **pathconf** command. While the **finfo** and **ffinfo** subroutines can be used to retrieve file information, it is preferred that programs use the **pathconf** interface.
FI_DIOCAP	When the **FI_DIOCAP** command is specified, the file's direct I/O capability information is returned. The buffer supplied by the application is of type **struct diocapbuf *** .

Return Values

Upon successful completion, the **finfo** and **ffinfo** subroutines return a value of 0 and the user supplied buffer is correctly filled in with the file information requested. If the **finfo** or **ffinfo** subroutines were unsuccessful, a value of **-1** is returned and the global **errno** variable is set to indicate the error.

Error Codes

Item	Description
EACCES	Search permission is denied for a component of the path prefix.
EINVAL	If the length specified for the user buffer is greater than **MAX_FINFO_BUF**. If the command argument is not supported. If **FI_DIOCAP** command is specified and the file object does not support Direct I/O.
ENAMETOOLONG	The length of the Path parameter string exceeds the **PATH_MAX** value.
ENOENT	The named file does not exist or the Path parameter points to an empty string.

Item	Description
ENOTDIR	A component of the path prefix is not a directory.
EBADF	File descriptor provided is not valid.

filter Subroutine

Purpose

Disables use of certain terminal capabilities.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
void filter(void);
```

Description

The **filter** subroutine changes the algorithm for initialising terminal capabilities that assume that the terminal has more than one line. A subsequent call to the **initscr** or **newterm** subroutine performs the following actions:

- Disables use of clear, cud, cud1, cup, cuu1, and vpa.
- Sets the value of the home string to the value of the cr. string.
- Sets lines equal to 1.

Any call to the **filter** subroutine must precede the call to the **initscr** or **newterm** subroutine.

flash Subroutine

Purpose

Flashes the screen.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
int flash(void);
```

Description

The **flash** subroutine alerts the user. It flashes the screen, or if that is not possible, it sounds the audible alarm on the terminal. If neither signal is possible, nothing happens.

Return Values

The **flash** subroutine always returns OK.

Examples

To cause the terminal to flash, enter:

```
flash();
```

flockfile, ftrylockfile, funlockfile Subroutine

Purpose

Provides for explicit application-level locking of stdio (FILE*) objects.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
void flockfile (FILE * file)
int ftrylockfile (FILE * file)
void funlockfile (FILE * file)
```

Description

The **flockfile**, **ftrylockfile** and **funlockfile** functions provide for explicit application-level locking of stdio (**FILE***) objects. These functions can be used by a thread to delineate a sequence of I/O statements that are to be executed as a unit.

The **flockfile** function is used by a thread to acquire ownership of a (**FILE***) object.

The **ftrylockfile** function is used by a thread to acquire ownership of a (**FILE***) object if the object is available; **ftrylockfile** is a non-blocking version of **flockfile**.

The **funlockfile** function is used to relinquish the ownership granted to the thread. The behavior is undefined if a thread other than the current owner calls the **funlockfile** function.

Logically, there is a lock count associated with each (**FILE***) object. This count is implicitly initialised to zero when the (**FILE***) object is created. The (**FILE***) object is unlocked when the count is zero. When the count is positive, a single thread owns the (**FILE***) object. When the **flockfile** function is called, if the count is zero or if the count is positive and the caller owns the (**FILE***) object, the count is incremented. Otherwise, the calling thread is suspended, waiting for the count to return to zero. Each call to **funlockfile** decrements the count. This allows matching calls to **flockfile** (or successful calls to **ftrylockfile**) and **funlockfile** to be nested.

All functions that reference (**FILE***) objects behave as if they use **flockfile** and **funlockfile** internally to obtain ownership of these (**FILE***) objects.

Return Values

None for **flockfile** and **funlockfile**. The function ftrylock returns zero for success and non-zero to indicate that the lock cannot be acquired.

Implementation Specifics

These subroutines are part of Base Operating System (BOS) subroutines.

Realtime applications may encounter priority inversion when using FILE locks. The problem occurs when a high priority thread locks a file that is about to be unlocked by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of 7434 ways, such as by having critical sections that are guarded by file locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

floor, floorf, floorl, floord32, floord64, floord128, nearest, trunc, itrunc, and uitrunc Subroutines

Purpose

The **floor** subroutine, **floorf** subroutine, **floorl** subroutine, **nearest** subroutine, **trunc** subroutine, **floord32** subroutine, **floord64** subroutine, and **floord128** subroutine, round floating-point numbers to floating-point integer values.

The **itrunc** subroutine and **uitrunc** subroutine round floating-point numbers to signed and unsigned integers, respectively.

Libraries

IEEE Math Library (**libm.a**) or System V Math Library (**libmsaa.a**) Standard C Library (**libc.a**) (separate syntax follows)

Syntax

```
#include <math.h>
```

```
double floor ( x)
double x;
```

```
float floorf (x)
float x;
```

```
long double floorl (x)
long double x;
```

```
_Decimal32 floord32(x)
_Decimal32 x;

_Decimal64 floord64(x)
_Decimal64 x;

_Decimal128 floord128(x)
_Decimal128 x;
```

```
double nearest (x)
double x;
```

```
double trunc (x)
double x;
```

Standard C Library (**libc.a**)

```
#include <stdlib.h>
#include <limits.h>
```

```
int itrunc (x)
double x;
```

```
unsigned int uitrunc (x)
double x;
```

Description

The **floor**, **floorf**, **floorl**, **floord32**, **floord64**, and **floord128** subroutines return the largest floating-point integer value that is not greater than the *x* parameter.

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

The **nearest** subroutine returns the nearest floating-point integer value to the *x* parameter. If *x* lies exactly halfway between the two nearest floating-point integer values, an even floating-point integer is returned.

The **trunc** subroutine returns the nearest floating-point integer value to the *x* parameter in the direction of 0. This is equivalent to truncating off the fraction bits of the *x* parameter.

Note: The default floating-point rounding mode is *round to nearest.* All C main programs begin with the rounding mode set to *round to nearest.*

The **itrunc** subroutine returns the nearest signed integer to the *x* parameter in the direction of 0. This is equivalent to truncating the fraction bits from the *x* parameter and then converting *x* to a signed integer.

The **uitrunc** subroutine returns the nearest unsigned integer to the *x* parameter in the direction of 0. This action is equivalent to truncating off the fraction bits of the *x* parameter and then converting *x* to an unsigned integer.

Note: Compile any routine that uses subroutines from the **libm.a** library with the **-lm** flag. To compile the `floor.c` file, for example, enter:

```
cc floor.c -lm
```

The **itrunc**, **uitrunc**, **trunc**, and **nearest** subroutines are not part of the ANSI C Library.

Parameters

Item	Description
x	Specifies a double-precision floating-point value. For the **floorl** subroutine, specifies a long double-precision floating-point value.

Item	Description
y	Specifies a double-precision floating-point value. For the **floorl** subroutine, specifies some long double-precision floating-point value.

Return Values

Upon successful completion, the **floor**, **floorf**, **floorl**, **floord32**, **floord64**, and **floord128** subroutines return the largest integral value that is not greater than *x*, expressed as a **double**, **float**, **long double**, **_Decimal32**, **_Decimal64**, or **_Decimal128**, as appropriate for the return type of the function.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

If the correct value would cause overflow, a range error occurs and the**floor**, **floorf**, **floorl**, **floord32**, **floord64**, and **floord128** subroutines return the value of the macro **-HUGE_VAL**, **-HUGE_VALF**, **-HUGE_VALL**, **-HUGE_VAL_D32**, **-HUGE_VAL_D64**, and **-HUGE_VAL_D128**, respectively.

Error Codes

The **itrunc** and **uitrunc** subroutines return the **INT_MAX** value if *x* is greater than or equal to the **INT_MAX** value and the **INT_MIN** value if *x* is equal to or less than the **INT_MIN** value. The **itrunc** subroutine returns the **INT_MIN** value if *x* is a Quiet NaN(not-a-number) or Silent NaN. The **uitrunc** subroutine returns 0 if *x* is a Quiet NaN or Silent NaN. (The **INT_MAX** and **INT_MIN** values are defined in the **limits.h** file.) The **uitrunc** subroutine **INT_MAX** if *x* is greater than **INT_MAX** and 0 if *x* is less than or equal 0.0

Files

Item	Description
float.h	Contains the ANSI C **FLT_ROUNDS** macro.

flushinp Subroutine

Purpose

Discards input.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int flushinp(void);
```

Description

The **flushinp** subroutine discards (flushes) any characters in the input buffers associated with the current screen.

Return Values

The **flushinp** subroutine always returns OK.

Examples

To flush all type-ahead characters typed by the user but not yet read by the program, enter:

```
flushinp();
```

fma, fmaf, fmal, and fmad128 Subroutines

Purpose

Floating-point multiply-add.

Syntax

```
#include <math.h>

double fma (x, y, z)
double x;
double y;
double z;

float fmaf (x, y, z)
float x;
float y;
float z;

long double fmal (x, y, z)
long double x;
long double y;
long double z;

_Decimal128 fmad128 (x, y, z)
_Decimal128 x;
_Decimal128 y;
_Decimal128 z;
```

Description

The **fma**, **fmaf**, **fmal**, and **fmad128** subroutines compute (*x* * *y*) + *z*, rounded as one ternary operation. They compute the value (as if) to infinite precision and round once to the result format, according to the rounding mode characterized by the value of FLT_ROUNDS.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be multiplied by the *y* parameter.
y	Specifies the value to be multiplied by the *x* parameter.
z	Specifies the value to be added to the product of the *x* and *y* parameters.

Return Values

Upon successful completion, the **fma**, **fmaf**, **fmal**, and **fmad128** subroutines return (*x* * *y*) + *z*, rounded as one ternary operation.

If *x* or *y* are NaN, a NaN is returned.

If *x* multiplied by *y* is an exact infinity and *z* is also an infinity but with the opposite sign, a domain error occurs, and a NaN is returned.

If one of the *x* and *y* parameters is infinite, the other is zero, and the *z* parameter is not a NaN, a domain error occurs, and a NaN is returned.

If one of the *x* and *y* parameters is infinite, the other is zero, and *z* is a NaN, a NaN is returned and a domain error may occur.

If *x***y* is not 0*Inf nor Inf*0 and *z* is a NaN, a NaN is returned.

fmax, fmaxf, fmaxl, fmaxd32, fmaxd64, and fmaxd128 Subroutines

Purpose

Determines the maximum numeric value of two floating-point numbers.

Syntax

```
#include <math.h>

double fmax (x, y)
double x;
double y;

float fmaxf (x, y)
float x;
float y;

long double fmaxl (x, y)
long double x;
long double y;

_Decimal32 fmaxd32 (x, y);
_Decimal32 x;
_Decimal32 y;

_Decimal64 fmaxd64 (x, y);
_Decimal64 x;
_Decimal64 y;

_Decimal128 fmaxd128 (x, y);
_Decimal128 x;
_Decimal128 y;
```

Description

The **fmax**, **fmaxf**, **fmaxl**, **fmaxd32**, **fmaxd64**, and **fmaxd128** subroutines determine the maximum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, the **fmax**, **fmaxf**, **fmaxl**, **fmaxd32**, **fmaxd64**, and **fmaxd128** subroutines choose the numeric value.

Parameters

Item	Description
x	Specifies the value to be computed.
y	Specifies the value to be computed.

Return Values

Upon successful completion, the **fmaxl**, **fmaxf**, **fmaxl**, **fmaxd32**, **fmaxd64**, and **fmaxd128** subroutines return the maximum numeric value of their arguments.

If one argument is a NaN, the other argument is returned.

If *x* and *y* are NaN, a NaN is returned.

fmemopen Subroutine

Purpose

Opens a memory buffer stream.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
FILE *fmemopen (void *restrict buf, size_t size, const char *restrict mode);
```

Description

The **fmemopen** subroutine associates the buffer given by the *buf* and *size* arguments with a stream. The *buf* argument must be either a null pointer or point to a buffer that contains the value specified by the *size* parameter in bytes.

The *mode* argument is a character string having one of the following values:

- *r* or *rb* to open the stream for reading.
- *w* or *wb* to open the stream for writing.
- *a* or *ab* Append to pen the stream for writing at the first null byte.
- *r+* or *rb+* or *r+b* to open the stream for update (reading and writing).
- *w+* or *wb+* or *w+b* to open the stream for update (reading and writing). Truncates the buffer contents.
- *a+* or *ab+* or *a+b* Append to open the stream for update (reading and writing) and the initial position is at the first null byte.

The character *b* does not have any effect.

If a null pointer is specified as the *buf* argument, the **fmemopen** subroutine allocates the number of bytes specified by the *size* parameter to the memory by a call to the **malloc** subroutine. This buffer is automatically released when the stream is closed. Because this feature is only useful when the stream is opened for updating since there is no way to get a pointer to the buffer, the **fmemopen** subroutine call fails if the *mode* argument does not include a **+** character.

The stream maintains a current position in the buffer. This position is initially set to either the beginning of the buffer (for *r* and *w* modes) or to the first null byte in the buffer (for *a* modes). If no null byte is found in the append mode, the initial position is set to one byte after the end of the buffer.

If *buf* is a null pointer, the initial position is always set to the beginning of the buffer.

The stream also maintains the size of the current buffer contents. For modes *r* and *r+* the size is set to the value given by the *size* argument. For modes *w* and *w+* the initial size is zero and for modes *a* and *a+* the initial size is either the position of the first null byte in the buffer or the value of the size argument if no null byte is found.

A read operation on the stream does not advance the current buffer position behind the current buffer size. Reaching the buffer size in a read operation counts as end of file. Null bytes in the buffer have no special meaning for reads. The read operation starts at the current buffer position of the stream.

A write operation starts either at the current position of the stream (if mode does not contain *a* as the first character) or at the current size of the stream (if mode does not contain *a* as the first character). If the current position at the end of the write is larger than the current buffer size, the current buffer size is set to the current position. A write operation on the stream doesl not advance the current buffer size behind the size given in the size argument.

When a stream opened for writing is flushed or closed, a null byte is written at the current position or at the end of the buffer, depending on the size of the contents. If a stream open for update is flushed or closed and the last write has advanced the current buffer size, a null byte is written at the end of the buffer if it fits.

An attempt to seek a memory buffer stream to a negative position or to a position larger than the buffer size given in the size argument fails.

Return Values

Upon successful completion, the **fmemopen** subroutine returns a pointer to the object controlling the stream. Otherwise, a null pointer is returned, and the *errno* variable is set to indicate the error.

Error Codes

The **fmemopen** function returns the following error code:

Table 1. Error codes

Item	Description
EINVAL	The *size* argument specifies a buffer size of zero or the value of the *mode* argument is not valid or the *buf* argument is a null pointer and the *mode* argument does not include a **+** character.
EMFILE	**FOPEN_MAX** streams are currently open in the calling process.
ENOMEM	The *buf* argument is a null pointer and the allocation of a buffer of length specified by the *size* parameter has failed.

Examples

```
#include <stdio.h>
static char buffer[] = "foobar";
int
main (void)
{
int ch;
FILE *stream;
stream = fmemopen(buffer, strlen (buffer), "r");
if (stream == NULL)
/* handle error */;
while ((ch = fgetc(stream)) != EOF)
printf("Got %c\n", ch);
fclose(stream);
return (0);
}
```

The above program produces the following output:

Got f

Got o

Got o

Got b

Got a

Got r

fminf, fminl, fmind32, fmind64, and fmind128 Subroutines

Purpose

Determines the minimum numeric value of two floating-point numbers.

Syntax

```
#include <math.h>

float fminf (x, y)
float x;
float y;

long double fminl (x, y)
long double x;
long double y;

_Decimal32 fmind32 (x, y)
_Decimal32 x;
_Decimal32 y;

_Decimal64 fmind64 (x, y)
_Decimal64 x;
_Decimal64 y;

_Decimal128 fmind128 (x, y)
_Decimal128 x;
_Decimal128 y;
```

Description

The **fminf**, **fminl**, **fmind32**, **fmind64**, and **fmind128** subroutines determine the minimum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, the **fminf**, **fminl**, **fmind32**, **fmind64**, and **fmind128** subroutines choose the numeric value.

Parameters

Item	Description
x	Specifies the value to be computed.
y	Specifies the value to be computed.

Return Values

Upon successful completion, the **fminf**, **fminl**, **fmind32**, **fmind64**, and **fmind128** subroutines return the minimum numeric value of their arguments.

If one argument is a NaN, the other argument is returned.

If *x* and *y* are NaN, a NaN is returned.

fmod, fmodf, fmodl, fmodd32, fmodd64, and fmodd128 Subroutines

Purpose

Computes the floating-point remainder value.

Syntax

```
#include <math.h>

float fmodf (x, y)
float x;
float y;

long double fmodl (x, y)
long double x, y;

double fmod (x, y)
double x, y;
_Decimal32 fmodd32 (x, y)
_Decimal32 x, y;

_Decimal64 fmodd64 (x, y)
_Decimal64 x, y;

_Decimal128 fmodd128 (x, y)
_Decimal128 x, y;
```

Description

The **fmodf**, **fmodl**, **fmod**, **fmodd32**, **fmodd64**, and **fmodd128** subroutines return the floating-point remainder of the division of *x* by *y*.

An application that wants to check for error situations must set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. On return, if **errno** is the value of non-zero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is the value of non-zero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.
y	Specifies the value to be computed.

Return Values

The **fmodf**, **fmodl**, **fmod**, **fmodd32**, **fmodd64**, and **fmodd128** subroutines return the value *x*- *i* **y*. For the integer *i* such that, if *y* is nonzero, the result has the same sign as *x* and the magnitude is less than the magnitude of *y*.

If the correct value will cause underflow, and is not representable, a range error might occur, and 0.0 is returned.

If *x* or *y* is NaN, a NaN is returned.

If *y* is zero, a domain error occurs, and a NaN is returned.

If *x* is infinite, a domain error occurs, and a NaN is returned.

If *x* is ±0 and *y* is not zero, ±0 is returned.

If *x* is not infinite and *y* is ±Inf, *x* is returned.

If the correct value will cause underflow, and is representable, a range error might occur and the correct value is returned.

If the correct value is zero, rounding error might cause the return value to differ from 0.0. Depending on the values of *x* and *y*, and the rounding mode, the magnitude of the return value in this case might be near 0.0 or near the magnitude of y. This case can be avoided by using the decimal floating-point subroutines (**fmodd32**, **fmodd64**, and **fmodd128**).

fmtmsg Subroutine

Purpose

Display a message in the specified format on standard error, the console, or both.

Library

Standard C Library **(libc.a)**

Syntax

```
#include <fmtmsg.h>

int fmtmsg (long Classification,
const char *Label,
int Severity,
cont char *Text;
cont char *Action,
cont char *Tag)
```

Description

The **fmtmsg** subroutine can be used to display messages in a specified format instead of the traditional **printf** subroutine interface.

Base on a message's classification component, the **fmtmsg** subroutine either writes a formatted message to standard error, the console, or both.

A formatted message consists of up to five parameters. The *Classification* parameter is not part of a message displayed to the user, but defines the source of the message and directs the display of the formatted message.

Parameters

Item	Description
Classification	Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and system console). **major classifications** Identifies the source of the condition. Identifiers are: **MM_HARD** (hardware), **MM_SOFT** (software), and **MM_FIRM** (firmware). **message source subclassifications** Identifies the type of software in which the problem is detected. Identifiers are: **MM_APPL** (application), **MM_UTIL** (utility), and **MM_OPSYS** (operating system). **display subclassification** Indicates where the message is to be displayed. Identifiers are: **MM_PRINT** to display the message on the standard error stream, **MM_CONSOLE** to display the message on the system console. One or both identifiers may be used. **status subclassifications** Indicates whether the application will recover from the condition. Identifiers are:**MM_RECOVER** (recoverable) and **MM_RECOV** (non-recoverable). An additional identifier, **MM_NULLMC**, identifies that no classification component is supplied for the message.
Label	Identifies the source to the message. The format is two fields separated by a colon. The first field is up to 10 bytes, the second field is up to 14 bytes.
Severity	
Text	Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is null then a message will be issued stating that no text has been provided.
Action	Describes the first step to be taken in the error-recovery process. The **fmtmsg** subroutine precedes the action string with the prefix: `TO FIX:`. The *Action* string is not limited to a specific size.
Tag	An identifier which references online documentation for the message. Suggested usage is that *tag* includes the *Label* and a unique identifying number. A sample *tag* is `UX:cat:146`.

Environment Variables

The **MSGVERB** (message verbosity) environment variable controls the behavior of the **fmtmsg** subroutine.

MSGVERB tells the **fmtmsg** subroutine which message components it is to select when writing messages to standard error. The value of **MSGVERB** is a colon-separated list of optional keywords. **MSGVERB** can be set as follows:

```
MSGVERB=[keyword[:keyword[:...]]]
export  MSGVERB
```

Valid keywords are: *Label, Severity, Text, Action,* and *Tag*. If **MSGVERB** contains a keyword for a component and the component's value is not the component's null value, **fmtmsg** subroutine includes that component in the message when writing the message to standard error. If **MSGVERB** does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If **MSGVERB** is not defined, if its value is the null string, if its value is not of the correct format, of if it contains keywords other than the valid ones listed previously, the **fmtmsg** subroutine selects all components.

MSGVERB affects only which components are selected for display to standard error. All message components are included in console messages.

Application Usage

One or more message components may be systematically omitted from messages generated by an application by using the null value of the parameter for that component. The table below indicates the null values and identifiers for **fmtmsg** subroutine parameters. The parameters are of type `char*` unless otherwise indicated.

Parameter	Null-Value	Identifier
label	(char*)0	MM_NULLLBL
severity (type `int`)	0	MM_NULLSEV
class (type `long`)	0L	MM_NULLMC
text	(char*)0	MM_NULLTXT
action	(char*)0	MM_NULLACT
tag	(char*)0	MM_NULLTAG

Another means of systematically omitting a component is by omitting the component keywords when defining the MSGVERB environment variable.

Return Values

The exit codes for the **fmtmsg** subroutine are the following:

Item	Description
MM_OK	The function succeeded.
MM_NOTOK	The function failed completely.
MM_MOMSG	The function was unable to generate a message on standard error.
MM_NOCON	The function was unable to generate a console message.

Examples

1. The following example of the **fmtmsg** subroutine:

```
fmtmsg(MM_PRINT, "UX:cat", MM_ERROR, "illegal option",
"refer tp cat in user's reference manual", "UX:cat:001")
```

   produces a complete message in the specified message format:

```
UX:cat ERROR: illegal option
TO FIX: refer to cat in user's reference manual UX:cat:001
```

2. When the environment variable MSGVERB is set as follows:

```
MSGVERB=severity:text:action
```

and the Example 1 is used, the **fmtmsg** subroutine produces:

```
ERROR: illegal option
TO FIX: refer to cat in user's reference manual UX:cat:001
```

fnmatch Subroutine

Purpose

Matches file name patterns.

Library

Standard C Library (**libc. a**)

Syntax

```
#include <fnmatch.h>
```

```
int fnmatch ( Pattern,  String,  Flags);
int Flags;
const char *Pattern, *String;
```

Description

The **fnmatch** subroutine checks the string specified by the *String* parameter to see if it matches the pattern specified by the *Pattern* parameter.

The **fnmatch** subroutine can be used by an application or command that needs to read a dictionary and apply a pattern against each entry; the **find**command is an example of this. It can also be used by the **pax** command to process its *Pattern* variables, or by applications that need to match strings in a similar manner.

Parameters

Item	Description
Pattern	Contains the pattern to which the *String* parameter is to be compared. The *Pattern* parameter can include the following special characters: *** (asterisk)** Matches zero, one, or more characters. **? (question mark)** Matches any single character, but will not match 0 (zero) characters. **[ ] (brackets)** Matches any one of the characters enclosed within the brackets. If a pair of characters separated by a dash are contained within the brackets, the pattern matches any character that lexically falls between the two characters in the current locale.
String	Contains the string to be compared against the *Pattern* parameter.

Item	Description
Flags	Contains a bit flag specifying the configurable attributes of the comparison to be performed by the **fnmatch** subroutine. The *Flags* parameter modifies the interpretation of the *Pattern* and *String* parameters. It is the bitwise inclusive OR of zero or more of the following flags (defined in the **fnmatch.h** file): **FNM_PATHNAME** Indicates the / (slash) in the *String* parameter matches a / in the *Pattern* parameter. **FNM_PERIOD** Indicates a leading period in the *String* parameter matches a period in the *Pattern* parameter. **FNM_NOESCAPE** Enables quoting of special characters using the \ (backslash).

If the **FNM_ PATHNAME** flag is set in the *Flags* parameter, a / (slash) in the *String* parameter is explicitly matched by a / in the *Pattern* parameter. It is not matched by either the * (asterisk) or ? (question-mark) special characters, nor by a bracket expression. If the **FNM_PATHNAME** flag is not set, the / is treated as an ordinary character.

If the **FNM_PERIOD** flag is set in the *Flags* parameter, then a leading period in the *String* parameter only matches a period in the *Pattern* parameter; it is not matched by either the asterisk or question-mark special characters, nor by a bracket expression. The setting of the **FNM_PATHNAME** flag determines a period to be leading, according to the following rules:

- If the **FNM_PATHNAME** flag is set, a . (period) is leading only if it is the first character in the *String* parameter or if it immediately follows a /.
- If the **FNM_PATHNAME** flag is not set, a . (period) is leading only if it is the first character of the *String* parameter. If **FNM_PERIOD** is not set, no special restrictions are placed on matching a period.

If the **FNM_NOESCAPE** flag is not set in the *Flags* parameter, a \ (backslash) character in the *Pattern* parameter, followed by any other character, will match that second character in the *String* parameter. For example, \\ will match a backslash in the *String* parameter. If the **FNM_NOESCAPE** flag is set, a \ (backslash) will be treated as an ordinary character.

Return Values

If the value in the *String* parameter matches the pattern specified by the *Pattern* parameter, the **fnmatch** subroutine returns 0. If there is no match, the **fnmatch** subroutine returns the **FNM_NOMATCH** constant, which is defined in the **fnmatch.h** file. If an error occurs, the **fnmatch** subroutine returns a nonzero value.

Files

Item	Description
/usr/include/fnmatch.h	Contains system-defined flags and constants.

fopen, fopen64, freopen, freopen64, fopen_s or fdopen Subroutine

Purpose

Opens a stream and handles runtime constraint violations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
#define STDC_WANT_LIB_EXT1 1
FILE  *fopen ( Path,  Type)
const char *Path, *Type;

FILE  *fopen64 ( Path,  Type)
char  *Path, *Type;

FILE  *freopen (Path, Type,  Stream)
const char *Path, *Type;
FILE  *Stream;

FILE  *freopen64 (Path, Type,  Stream)
char  *Path, *Type;
FILE  *Stream;

FILE  *fdopen ( FileDescriptor, Type)
int   FileDescriptor;
const char *Type;

errno_t fopen_s ( streamptr, filename, mode)
FILE * *streamptr ;
const char * filename ;
const char * mode ;
```

Description

The **fopen** and **fopen64** subroutines open the file named by the *Path* parameter and associate a stream with it and return a pointer to the **FILE** structure of this stream.

When you open a file for update, you can perform both input and output operations on the resulting stream. However, an output operation cannot be directly followed by an input operation without an intervening **fflush** subroutine call or a file positioning operation (**fseek**, **fseeko**, **fseeko64**, **fsetpos**, **fsetpos64** or **rewind** subroutine). Also, an input operation cannot be directly followed by an output operation without an intervening flush or file positioning operation, unless the input operation encounters the end of the file.

When you open a file for appending (that is, when the *Type* parameter is set to **a**), it is impossible to overwrite information already in the file.

If two separate processes open the same file for append, each process can write freely to the file without destroying the output being written by the other. The output from the two processes is intermixed in the order in which it is written to the file.

Note: If the data is buffered, it is not actually written until it is flushed.

The **freopen** and **freopen64** subroutines first attempt to flush the stream and close any file descriptor associated with the *Stream* parameter. Failure to flush the stream or close the file descriptor is ignored.

The **freopen** and **freopen64** subroutines substitute the named file in place of the open stream. The original stream is closed regardless of whether the subsequent open succeeds. The **freopen** and **freopen64** subroutines returns a pointer to the **FILE** structure associated with the *Stream* parameter. The **freopen** and **freopen64** subroutines is typically used to attach the pre-opened streams associated with standard input (**stdin**), standard output (**stdout**), and standard error (**stderr**) streams to other files.

The **fdopen** subroutine associates a stream with a file descriptor obtained from an **openx** subroutine, **dup** subroutine, **creat** subroutine, or **pipe** subroutine. These subroutines open files but do not return pointers to **FILE** structures. Many of the standard I/O package subroutines require pointers to **FILE** structures.

The *Type* parameter for the **fdopen** subroutine specifies the mode of the stream, such as **r** to open a file for reading, or **a** to open a file for appending (writing at the end of the file). The mode value of the *Type* parameter specified with the **fdopen** subroutine must agree with the mode of the file specified when the file was originally opened or created.

Note: Using the **fdopen** subroutine with a file descriptor obtained from a call to the **shm_open** subroutine must be avoided and might result in an error on the next **fread**, **fwrite** or **fflush** call.

The largest value that can be represented correctly in an object of type off_t will be established as the offset maximum in the open file description.

The **fopen_s** subroutine opens the file by using the name of the string pointed to by the **filename** parameter, and associates a stream with the file.

Files are opened for writing with exclusive (also known as non shared) access. If the file is created, and the first character of the **mode** parameter is not **u**, and if the underlying system supports exclusive mode concept, the file has a permission that prevents other users on the system from accessing the file.

If the file is created and the first character of the **mode** parameter is **u**, the file retains the system default file access permissions until the file is closed.

If the file is opened successfully, the pointer to the **FILE** structure that is pointed to by the *streamptr* parameter is set to the pointer that points to the object controlling the opened file. Otherwise, the pointer to the **FILE** structure pointed to by the *streamptr* parameter is set to a null pointer, and the file retains the system default file access permissions until the file is closed.

Runtime Constraints

1. For the **fopen_s** subroutine, the *streamptr*, *filename* or *mode* parameters must not be a null pointer.
2. If there is a runtime constraint violation, the **fopen_s** subroutine does not attempt to open a file. If the *streamptr* parameter is not a null pointer, the **fopen_s** subroutine sets the *streamptr* parameter to the null pointer.

Parameters

Item	Description
Path	Points to a character string that contains the name of the file to be opened.

Item	Description
Type	Points to a character string that has one of the following values: **r** Opens a text file for reading. **w** Creates a new text file for writing, or opens and truncates a file to 0 length. **a** Appends (opens a text file for writing at the end of the file, or creates a file for writing). **rb** Opens a binary file for reading. **wb** Creates a binary file for writing, or opens and truncates a file to 0. **ab** Appends (opens a binary file for writing at the end of the file, or creates a file for writing). **r+** Opens a file for update (reading and writing). **w+** Truncates or creates a file for update. **a+** Appends (opens a text file for writing at end of file, or creates a file for writing). **r+b , rb+** Opens a binary file for update (reading and writing). **w+b , wb+** Creates a binary file for update, or opens and truncates a file to 0 length. **a+b , ab+** Appends (opens a binary file for update, writing at the end of the file, or creates a file for writing). **wx** Creates a text file for writing. **wbx** Creates a binary file for writing. **w+x** Creates a text file for updating. **w+bx or wb+x** Creates a binary file for updating. **Note:** • The operating system does not distinguish between text and binary files. • The **b** value in the *Type* parameter is ignored. • Opening a file with exclusive mode (**x** as the last character in the mode argument) fails, if the file already exists or cannot be created. Otherwise, the file is created with exclusive (also known as non shared) access if the underlying system supports exclusive access. • The **fdopen** subroutine has no impact on exclusive mode.
Stream	Specifies the input stream.
FileDescriptor	Specifies a valid open file descriptor.

Item	Description
streamptr	Specifies the stream that is associated with the file name, and the value cannot be null.
filename	Specifies the file name to be opened, and the value cannot be null.
mode	The value cannot be null. The mode parameter is the same as the **Type** parameter described for **fopen** subroutine, with the addition that the modes starting with the character **w** or **a** can be preceded by the character **u** as shown below: **uw** Truncates to 0 or creates a text file for writing and has default permissions. **uwx** Creates a text file for writing and has default permissions. **ua** Opens or creates a text file for writing at the end of the file and has default permissions. **uwb** Truncates to 0 or creates a binary file for writing and has default permissions. **uwbx** Creates a binary file for writing and has default permissions. **uab** Opens or creates a binary file for writing at the end of the file and has default permissions. **uw+** Truncates to 0 or creates a text file for update and has default permissions. **uw+x** Creates a text file for update and has default permissions. **ua+ append** Opens or creates a text file for update and writing at the end-of-file and has default permissions. **uw+b or uwb+** Truncates to 0 or creates a binary file for update and has default permissions. **uw+bx or uwb+x** Creates a binary file for update and has default permissions. **ua+b or uab+ append** Opens or creates a binary file for update and writing at the end-of-file and has default permissions. **Note:** If the **mode** parameter is not preceded with **u**, the file permissions are user only.

Return Values

If the **fdopen**, **fopen**, **fopen64**, **freopen** or **freopen64** subroutine is unsuccessful, a null pointer is returned and the **errno** global variable is set to indicate the error.

The **fopen_s** subroutine returns a zero if it opens the file. If the file is not opened or if there is a runtime constraint violation, the **fopen_s** subroutine returns a nonzero value.

Error Codes

The **fopen**, **fopen64**, **freopen** and **freopen64** subroutines are unsuccessful if the following is true:

Item	Description
EACCES	Search permission is denied on a component of the path prefix, the file exists and the permissions specified by the mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
ELOOP	Too many symbolic links were encountered in resolving path.
EINTR	A signal was received during the process.
EISDIR	The named file is a directory and the process does not have write access to it.
ENAMETOOLONG	The length of the filename exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.
EMFILE	The maximum number of files allowed are currently open.
ENOENT	The named file does not exist or the *File Descriptor* parameter points to an empty string.
ENOSPC	The file is not yet created and the directory or file system to contain the new file cannot be expanded.
ENOTDIR	A component of the path prefix is not a directory.
ENXIO	The named file is a character- or block-special file, and the device associated with this special file does not exist.
EOVERFLOW	The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t.
EROFS	The named file resides on a read-only file system and does not have write access.
ETXTBSY	The file is a pure-procedure (shared-text) file that is being executed and the process does not have write access.

The **fdopen**, **fopen**, **fopen64**, **freopen** and **freopen64** subroutines are unsuccessful if the following is true:

Item	Description
EINVAL	The value of the *Type* argument is not valid.
EINVAL	The value of the *mode* argument is not valid.
EMFILE	**FOPEN_MAX** streams are currently open in the calling process.
EMFILE	**STREAM_MAX** streams are currently open in the calling process.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.
ENOMEM	Insufficient storage space is available.

The **freopen** and **fopen** subroutines are unsuccessful if the following is true:

Item	Description
EOVERFLOW	The named file is a size larger than 2 Gigabytes.

The **fdopen** subroutine is unsuccessful if the following is true:

Item	Description
EBADF	The value of the *File Descriptor* parameter is not valid.

POSIX

Item	Description
w	Truncates to 0 length or creates text file for writing.
w+	Truncates to 0 length or creates text file for update.
a	Opens or creates text file for writing at end of file.
a+	Opens or creates text file for update, writing at end of file.

SAA

At least eight streams, including three standard text streams, can open simultaneously. Both binary and text modes are supported.

fork, f_fork, or vfork Subroutine

Purpose

Creates a new process.

Libraries

fork, **f_fork**, and **vfork**: Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
pid_t fork(void)
```

```
pid_t f_fork(void)
```

```
int vfork(void)
```

Description

The **fork** subroutine creates a new process. The new process (child process) is an almost exact copy of the calling process (parent process). The child process inherits the following attributes from the parent process:

- Environment
- Close-on-exec flags (described in the **exec** subroutine)
- Signal handling settings (such as the **SIG_DFL** value, the **SIG_IGN** value, and the *Function Address* parameter)
- Set user ID mode bit
- Set group ID mode bit
- Profiling on and off status
- Nice value
- All attached shared libraries

- Process group ID
- **tty** group ID (described in the **exit**, **atexit**, or **_exit** subroutine, **signal** subroutine, and **raise** subroutine)
- Current directory
- Root directory
- File-mode creation mask (described in the **umask** subroutine)
- File size limit (described in the **ulimit** subroutine)
- Attached shared memory segments (described in the **shmat** subroutine)
- Attached mapped file segments (described in the **shmat** subroutine)
- Debugger process ID and multiprocess flag if the parent process has multiprocess debugging enabled (described in the **ptrace** subroutine).

The child process differs from the parent process in the following ways:

- The child process has only one user thread; it is the one that called the **fork** subroutine.
- The child process has a unique process ID.
- The child process ID does not match any active process group ID.
- The child process has a different parent process ID.
- The child process has its own copy of the file descriptors for the parent process. However, each file descriptor of the child process shares a common file pointer with the corresponding file descriptor of the parent process.
- All **semadj** values are cleared. For information about **semadj** values, see the **semop** subroutine.
- Process locks, text locks, and data locks are not inherited by the child process. For information about locks, see the **plock** subroutine.
- If multiprocess debugging is turned on, the **trace** flags are inherited from the parent; otherwise, the **trace** flags are reset. For information about request 0, see the **ptrace** subroutine.
- The child process **utime**, **stime**, **cutime**, and **cstime** subroutines are set to 0. (For more information, see the **getrusage** , **times**, and **vtimes** subroutines.)
- Any pending alarms are cleared in the child process. (For more information, see the **incinterval**, **setitimer**, and **alarm** subroutines.)
- The set of signals pending for the child process is initialized to an empty set.
- The child process can have its own copy of the message catalogue for the parent process.

Attention: If you are using the **fork** or **vfork** subroutines with an X Window System, X Toolkit, or Motif application, open a separate display connection (socket) for the forked process. If the child process uses the same display connection as the parent, the X Server will not be able to interpret the resulting data.

The **f_fork** subroutine is similar to **fork**, except for:

- It is required that the child process calls one of the **exec** functions immediately after it is created. Since the **fork** handlers are never called, the application data, mutexes and the locks are all undefined in the child process.

The **vfork** subroutine is supported as a compatibility interface for older Berkeley Software Distribution (BSD) system programs and can be used by compiling with the Berkeley Compatibility Library (**libbsd.a**).

In the Version 4 of the operating system, the parent process does not have to wait until the child either exits or executes, as it does in BSD systems. The child process is given a new address space, as in the **fork** subroutine. The child process does not share any parent address space.

Attention: When using the **fork** or **vfork** subroutines with an Enhanced X-Windows, X Toolkit, or Motif application, a separate display connection (socket) should be opened for the forked process. The child process should never use the same display connection as the parent. Display connections are embodied with sockets, and sockets are inherited by the child process. Any attempt to have multiple processes writing to the same display connection results in the random

interleaving of X protocol packets at the word level. The resulting data written to the socket will not be valid or undefined X protocol packets, and the X Server will not be able to interpret it.

Attention: Although the **fork** and **vfork** subroutine may be used with Graphics Library applications, the child process must not make any additional Graphics Library subroutine calls. The child application inherits some, but not all of the graphics hardware resources of the parent. Drawing by the child process may hang the graphics adapter, the Enhanced X Server, or may cause unpredictable results and place the system into an unpredictable state.

For additional information, see the **/usr/lpp/GL/README** file.

Return Values

Upon successful completion, the **fork** subroutine returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the **errno** global variable is set to indicate the error.

Error Codes

The **fork** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EAGAIN	Exceeds the limit on the total number of processes running either systemwide or by a single user, or the system does not have the resources necessary to create another process.
ENOMEM	Not enough space exists for this process.
EPROCLIM	If WLM is running, the limit on the number of processes or threads in the class may have been met.

fp_any_enable, fp_is_enabled, fp_enable_all, fp_enable, fp_disable_all, or fp_disable Subroutine

Purpose

These subroutines allow operations on the floating-point trap control.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fptrap.h>
```

```
int fp_any_enable()
int fp_is_enabled( Mask)
fptrap_t Mask;
```

```
void fp_enable_all()
void fp_enable(Mask)
fptrap_t Mask;
```

```
void fp_disable_all()
void fp_disable(Mask)
fptrap_t Mask;
```

Description

Floating point traps must be enabled before traps can be generated. These subroutines aid in manipulating floating-point traps and identifying the trap state and type.

In order to take traps on floating point exceptions, the **fp_trap** subroutine must first be called to put the process in serialized state, and the **fp_enable** subroutine or **fp_enable_all** subroutine must be called to enable the appropriate traps.

The header file **fptrap.h** defines the following names for the individual bits in the floating-point trap control:

Item	Description
TRP_INVALID	Invalid Operation Summary
TRP_DIV_BY_ZERO	Divide by Zero
TRP_OVERFLOW	Overflow
TRP_UNDERFLOW	Underflow
TRP_INEXACT	Inexact Result

Parameters

Item	Description
Mask	A 32-bit pattern that identifies floating-point traps.

Return Values

The **fp_any_enable** subroutine returns 1 if any floating-point traps are enabled. Otherwise, 0 is returned.

The **fp_is_enabled** subroutine returns 1 if the floating-point traps specified by the *Mask* parameter are enabled. Otherwise, 0 is returned.

The **fp_enable_all** subroutine enables all floating-point traps.

The **fp_enable** subroutine enables all floating-point traps specified by the *Mask* parameter.

The **fp_disable_all** subroutine disables all floating-point traps.

The **fp_disable** subroutine disables all floating-point traps specified by the *Mask* parameter.

fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag Subroutine

Purpose

Allows operations on the floating-point exception flags.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <float.h>
#include <fpxcp.h>
```

```
void fp_clr_flag( Mask)
fpflag_t Mask;
```

```
void fp_set_flag(Mask)
fpflag_t Mask;
```

```
fpflag_t fp_read_flag( )
```

```
fpflag_t fp_swap_flag(Mask)
fpflag_t Mask;
```

Description

These subroutines aid in determining both when an exception has occurred and the exception type. These subroutines can be called explicitly around blocks of code that may cause a floating-point exception.

According to the *IEEE Standard for Binary Floating-Point Arithmetic*, the following types of floating-point operations must be signaled when detected in a floating-point operation:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact

An invalid operation occurs when the result cannot be represented (for example, a **sqrt** operation on a number less than 0).

The *IEEE Standard for Binary Floating-Point Arithmetic* states: "For each type of exception, the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time."

Floating-point operations can set flags in the floating-point exception status but cannot clear them. Users can clear a flag in the floating-point exception status using an explicit software action such as the **fp_swap_flag (**0**)** subroutine.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

Item	Description
FP_INVALID	Invalid operation summary
FP_OVERFLOW	Overflow
FP_UNDERFLOW	Underflow
FP_DIV_BY_ZERO	Division by 0
FP_INEXACT	Inexact result

In addition to these flags, the operating system supports additional information about the cause of an invalid operation exception. The following flags also indicate floating-point exception status and defined in the **fpxcp.h** file. The flag number for each exception type varies, but the mnemonics are the same for all ports. The following invalid operation detail flags are not required for conformance to the IEEE floating-point exceptions standard:

Item	Description
FP_INV_SNAN	Signaling NaN
FP_INV_ISI	INF - INF

Item	Description
FP_INV_IDI	INF / INF
FP_INV_ZDZ	0 / 0
FP_INV_IMZ	INF x 0
FP_INV_CMP	Unordered compare
FP_INV_SQRT	Square root of a negative number
FP_INV_CVI	Conversion to integer error
FP_INV_VXSOFT	Software request

Parameters

Item	Description
Mask	A 32-bit pattern that identifies floating-point exception flags.

Return Values

The **fp_clr_flag** subroutine resets the exception status flags defined by the *Mask* parameter to 0 (false). The remaining flags in the exception status are unchanged.

The **fp_set_flag** subroutine sets the exception status flags defined by the *Mask* parameter to 1 (true). The remaining flags in the exception status are unchanged.

The **fp_read_flag** subroutine returns the current floating-point exception status. The flags in the returned exception status can be tested using the flag definitions above. You can test individual flags or sets of flags.

The **fp_swap_flag** subroutine writes the *Mask* parameter into the floating-point status and returns the floating-point exception status from before the write.

Users set or reset multiple exception flags using **fp_set_flag** and **fp_clr_flag** by ANDing or ORing definitions for individual flags. For example, the following resets both the overflow and inexact flags:

```
fp_clr_flag (FP_OVERFLOW | FP_INEXACT)
```

fp_cpusync Subroutine

Purpose

Queries or changes the floating-point exception enable (FE) bit in the Machine Status register (MSR).

Note: This subroutine has been replaced by the **fp_trapstate** subroutine. The **fp_cpusync** subroutine is supported for compatibility, but the **fp_trapstate** subroutine should be used for development.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fptrap.h>
```

```
int fp_cpusync ( Flag);
int Flag;
```

Description

The **fp_cpusync** subroutine is a service routine used to query, set, or reset the Machine Status Register (MSR) floating-point exception enable (FE) bit. The MSR FE bit determines whether a processor runs in pipeline or serial mode. Floating-point traps can only be generated by the hardware when the processor is in synchronous mode.

The **fp_cpusync** subroutine changes only the MSR FE bit. It is a service routine for use in developing custom floating-point exception-handling software. If you are using the **fp_enable** or **fp_enable_all** subroutine or the **fp_sh_trap_info** or **fp_sh_set_stat** subroutine, you must use the **fp_trap** subroutine to place the process in serial mode.

Parameters

Item	Description
Flag	Specifies to query or modify the MSR FE bit: **FP_SYNC_OFF** Sets the FE bit in the MSR to Off, which disables floating-point exception processing immediately. **FP_SYNC_ON** Sets the FE bit in the MSR to On, which enables floating-exception processing for the next floating-point operation. **FP_SYNC_QUERY** Returns the current state of the process (either **FP_SYNC_ON** or **FP_SYNC_OFF**) without modifying it.

If called with any other value, the **fp_cpusync** subroutine returns **FP_SYNC_ERROR**.

Return Values

If called with the **FP_SYNC_OFF** or **FP_SYNC_ON** flag, the **fp_cpusync** subroutine returns a value indicating which flag was in the previous state of the process.

If called with the **FP_SYNC _QUERY** flag, the **fp_cpusync** subroutine returns a value indicating the current state of the process, either the **FP_SYNC_OFF** or **FP_SYNC_ON** flag.

Error Codes

If the **fp_cpusync** subroutine is called with an invalid parameter, the subroutine returns **FP_SYNC_ERROR**. No other errors are reported.

fp_flush_imprecise Subroutine

Purpose

Forces imprecise signal delivery.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fptrap.h>
```

```
void fp_flush_imprecise ()
```

Description

The **fp_flush_imprecise** subroutine forces any imprecise interrupts to be reported. To ensure that no signals are lost when a program voluntarily exits, use this subroutine in combination with the **atexit** subroutine.

Example

The following example illustrates using the **atexit** subroutine to run the **fp_flush_imprecise** subroutine before a program exits:

```
#include <fptrap.h>
#include <stdlib.h>
#include <stdio.h>
  if (0!=atexit(fp_flush_imprecise))
           puts ("Failure in atexit(fp_flush_imprecise) ");
```

fp_invalid_op, fp_divbyzero, fp_overflow, fp_underflow, fp_inexact, fp_any_xcp Subroutine

Purpose

Tests to see if a floating-point exception has occurred.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <float.h>
#include <fpxcp.h>
```

```
int
fp_invalid_op()
int fp_divbyzero()
```

```
int fp_overflow()
int fp_underflow()
```

```
int
fp_inexact()
int fp_any_xcp()
```

Description

These subroutines aid in determining when an exception has occurred and the exception type. These subroutines can be called explicitly after blocks of code that may cause a floating-point exception.

Return Values

The **fp_invalid_op** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_divbyzero** subroutine returns a value of 1 if a floating-point divide-by-zero exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_overflow** subroutine returns a value of 1 if a floating-point overflow exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_underflow** subroutine returns a value of 1 if a floating-point underflow exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_inexact** subroutine returns a value of 1 if a floating-point inexact exception status flag is set. Otherwise, a value of 0 is returned.

The **fp_any_xcp** subroutine returns a value of 1 if a floating-point invalid operation, divide-by-zero, overflow, underflow, or inexact exception status flag is set. Otherwise, a value of 0 is returned.

fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf, fp_iop_zrdzr, fp_iop_infmzr, fp_iop_invcmp, fp_iop_sqrt, fp_iop_convert, or fp_iop_vxsoft Subroutines

Purpose

Tests to see if a floating-point exception has occurred.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <float.h>
#include <fpxcp.h>
```

```
int fp_iop_snan()
int fp_iop_infsinf()
```

```
int
fp_iop_infdinf()
int fp_iop_zrdzr()
```

```
int
fp_iop_infmzr()
int fp_iop_invcmp()
```

```
int
fp_iop_sqrt()
int fp_iop_convert()
```

```
int
fp_iop_vxsoft ();
```

Description

These subroutines aid in determining when an exception has occurred and the exception type. These subroutines can be called explicitly after blocks of code that may cause a floating-point exception.

Return Values

The **fp_iop_snan** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a signaling NaN (NaNS) flag. Otherwise, a value of 0 is returned.

The **fp_iop_infsinf** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF-INF flag. Otherwise, a value of 0 is returned.

The **fp_iop_infdinf** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF/INF flag. Otherwise, a value of 0 is returned.

The **fp_iop_zrdzr** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a 0.0/0.0 flag. Otherwise, a value of 0 is returned.

The **fp_iop_infmzr** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to an INF*0.0 flag. Otherwise, a value of 0 is returned.

The **fp_iop_invcmp** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to a compare involving a NaN. Otherwise, a value of 0 is returned.

The **fp_iop_sqrt** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to the calculation of a square root of a negative number. Otherwise, a value of 0 is returned.

The **fp_iop_convert** subroutine returns a value of 1 if a floating-point invalid-operation exception status flag is set due to the conversion of a floating-point number to an integer, where the floating-point number was a NaN, an INF, or was outside the range of the integer. Otherwise, a value of 0 is returned.

The **fp_iop_vxsoft** subroutine returns a value of 1 if the VXSOFT detail bit is on. Otherwise, a value of 0 is returned.

fp_raise_xcp Subroutine

Purpose

Generates a floating-point exception.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fpxcp.h>

int fp_raise_xcp( mask)
fpflag_t mask;
```

Description

The **fp_raise_xcp** subroutine causes any floating-point exceptions defined by the *mask* parameter to be raised immediately. If the exceptions defined by the *mask* parameter are enabled and the program is running in serial mode, the signal for floating-point exceptions, **SIGFPE**, is raised.

If more than one exception is included in the *mask* variable, the exceptions are raised in the following order:

1. Invalid
2. Dividebyzero
3. Underflow
4. Overflow
5. Inexact

Thus, if the user exception handler does not disable further exceptions, one call to the **fp_raise_xcp** subroutine can cause the exception handler to be entered many times.

Parameters

Item	Description
mask	Specifies a 32-bit pattern that identifies floating-point traps.

Return Values

The **fp_raise_xcp** subroutine returns 0 for normal completion and returns a nonzero value if an error occurs.

fp_read_rnd or fp_swap_rnd Subroutine

Purpose

Read and set the IEEE floating-point rounding mode.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <float.h>
```

```
fprnd_t fp_read_rnd()
fprnd_t fp_swap_rnd( RoundMode)
fprnd_t RoundMode;
```

Description

The **fp_read_rnd** subroutine returns the current rounding mode. The **fp_swap_rnd** subroutine changes the rounding mode to the *RoundMode* parameter and returns the value of the rounding mode before the change.

Floating-point rounding occurs when the infinitely precise result of a floating-point operation cannot be represented exactly in the destination floating-point format (such as double-precision format).

The *IEEE Standard for Binary Floating-Point Arithmetic* allows floating-point numbers to be rounded in four different ways: round toward zero, round to nearest, round toward +INF, and round toward -INF. Once a rounding mode is selected it affects all subsequent floating-point operations until another rounding mode is selected.

Note: The default floating-point rounding mode is round to nearest. All C main programs begin with the rounding mode set to round to nearest.

The encodings of the rounding modes are those defined in the *ANSI C Standard*. The **float.h** file contains definitions for the rounding modes. Below is the **float.h** definition, the *ANSI C Standard* value, and a description of each rounding mode.

float.h Definition	ANSI Value	Description
FP_RND_RZ	0	Round toward 0
FP_RND_RN	1	Round to nearest
FP_RND_RP	2	Round toward +INF
FP_RND_RM	3	Round toward -INF

The **fp_swap_rnd** subroutine can be used to swap rounding modes by saving the return value from **fp_swap_rnd**(*RoundMode*). This can be useful in functions that need to force a specific rounding mode for use during the function but wish to restore the caller's rounding mode on exit. Below is a code fragment that accomplishes this action:

```
save_mode = fp_swap_rnd (new_mode);
....desired code using new_mode
(void) fp_swap_rnd(save_mode); /*restore caller's mode*/
```

Parameters

Item	Description
RoundMode	Specifies one of the following modes: **FP_RND_RZ**, **FP_RND_RN**, **FP_RND_RP**, or **FP_RND_RM**.

fp_sh_info, fp_sh_trap_info, or fp_sh_set_stat Subroutine

Purpose

From within a floating-point signal handler, determines any floating-point exception that caused the trap in the process and changes the state of the Floating-Point Status and Control register (FPSCR) in the user process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fpxcp.h>
#include <fptrap.h>
#include <signal.h>
```

```
void fp_sh_info( scp,  fcp,  struct_size)
struct sigcontext *scp;
struct fp_sh_info *fcp;
size_t struct_size;
```

```
void fp_sh_trap_info( scp,  fcp)
struct sigcontext *scp;
struct fp_ctx *fcp;
```

```
void fp_sh_set_stat( scp,  fpscr)
struct sigcontext *scp;
fpstat_t fpscr;
```

Description

These subroutines are for use within a user-written signal handler. They return information about the process that was running at the time the signal occurred, and they update the Floating-Point Status and Control register for the process.

Note: The **fp_sh_trap_info** subroutine is maintained for compatibility only. It has been replaced by the **fp_sh_info** subroutine, which should be used for development.

These subroutines operate only on the state of the user process that was running at the time the signal was delivered. They read and write the **sigcontext** structure. They do not change the state of the signal handler process itself.

The state of the signal handler process can be modified by the **fp_any_enable**, **fp_is_enabled**, **fp_enable_all**, **fp_enable**, **fp_disable_all**, or **fp_disable** subroutine.

fp_sh_info

The **fp_sh_info** subroutine returns information about the process that caused the trap by means of a floating-point context (**fp_sh_info**) structure. This structure contains the following information:

```
typedef struct fp_sh_info {
fpstat_t       fpscr;
fpflag_t       trap;
short          trap_mode;
char           flags;
char           extra;
} fp_sh_info_t;
```

The fields are:

Item	Description
fpscr	The Floating-Point Status and Control register (FPSCR) in the user process at the time the interrupt occurred.
trap	A mask indicating the trap or traps that caused the signal handler to be entered. This mask is the logical OR operator of the enabled floating-point exceptions that occurred to cause the trap. This mask can have up to two exceptions; if there are two, the **INEXACT** signal must be one of them. If the mask is 0, the **SIGFPE** signal was raised not by a floating-point operation, but by the **kill** or **raise** subroutine or the **kill** command.
trap_mode	The trap mode in effect in the process at the time the signal handler was entered. The values returned in the **fp_sh_info.trap_mode** file use the following argument definitions: **FP_TRAP_OFF** Trapping off **FP_TRAP_SYNC** Precise trapping on **FP_TRAP_IMP_REC** Recoverable imprecise trapping on **FP_TRAP_IMP** Non-recoverable imprecise trapping on
flags	This field is interpreted as an array of bits and should be accessed with masks. The following mask is defined: **FP_IAR_STAT** If the value of the bit at this mask is 1, the exception was precise and the IAR points to the instruction that caused the exception. If the value bit at this mask is 0, the exception was imprecise.

fp_sh_trap_info

The **fp_sh_trap_info** subroutine is maintained for compatibility only. The **fp_sh_trap_info** subroutine returns information about the process that caused the trap by means of a floating-point context (**fp_ctx**) structure. This structure contains the following information:

```
fpstat_t fpscr;
fpflag_t trap;
```

The fields are:

Item	Description
fpscr	The Floating-Point Status and Control register (FPSCR) in the user process at the time the interrupt occurred.

Item	Description
`trap`	A mask indicating the trap or traps that caused the signal handler to be entered. This mask is the logical OR operator of the enabled floating-point exceptions that occurred to cause the trap. This mask can have up to two exceptions; if there are two, the **INEXACT** signal must be one of them. If the mask is 0, the **SIGFPE** signal was raised not by a floating-point operation, but by the **kill** or **raise** subroutine or the **kill** command.

fp_sh_set_stat

The **fp_sh_set_stat** subroutine updates the Floating-Point Status and Control register (FPSCR) in the user process with the value in the `fpscr` field.

The signal handler must either clear the exception bit that caused the trap to occur or disable the trap to prevent a recurrence. If the instruction generated more than one exception, and the signal handler clears only one of these exceptions, a signal is raised for the remaining exception when the next floating-point instruction is executed in the user process.

Parameters

Item	Description
fcp	Specifies a floating-point context structure.
scp	Specifies a **sigcontext** structure for the interrupt.
struct_size	Specifies the size of the **fp_sh_info** structure.
fpscr	Specifies which Floating-Point Status and Control register to update.

fp_trap Subroutine

Purpose

Queries or changes the mode of the user process to allow floating-point exceptions to generate traps.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fptrap.h>

int fp_trap( flag)
int flag;
```

Description

The **fp_trap** subroutine queries and changes the mode of the user process to allow or disallow floating-point exception trapping. Floating-point traps can only be generated when a process is executing in a traps-enabled mode.

The default state is to execute in pipelined mode and not to generate floating-point traps.

Note: The **fp_trap** routines only change the execution state of the process. To generate floating-point traps, you must also enable traps. Use the **fp_enable** and **fp_enable_all** subroutines to enable traps.

Before calling the **fp_trap(FP_TRAP_SYNC)** routine, previous floating-point operations can set to True certain exception bits in the Floating-Point Status and Control register (FPSCR). Enabling these

Cexceptions and calling the **fp_trap(FP_TRAP_SYNC)** routine does not cause an immediate trap to occur. That is, the operation of these traps is edge-sensitive, not level-sensitive.

The **fp_trap** subroutine does not clear the exception history. You can query this history by using any of the following subroutines:

- **fp_any_xcp**
- **fp_divbyzero**
- **fp_iop_convert**
- **fp_iop_infdinf**
- **fp_iop_infmzr**
- **fp_iop_infsinf**
- **fp_iop_invcmp**
- **fp_iop_snan**
- **fp_iop_sqrt**
- **fp_iop_vxsoft**
- **fp_iop_zrdzr**
- **fp_inexact**
- **fp_invalid_op**
- **fp_overflow**
- **fp_underflow**

Parameters

Item	Description
flag	Specifies a query of or change in the mode of the user process: **FP_TRAP_OFF** Puts the user process into trapping-off mode and returns the previous mode of the process, either **FP_TRAP_SYNC**, **FP_TRAP_IMP**, **FP_TRAP_IMP_REC**, or **FP_TRAP_OFF**. **FP_TRAP_QUERY** Returns the current mode of the user process. **FP_TRAP_SYNC** Puts the user process into precise trapping mode and returns the previous mode of the process. **FP_TRAP_IMP** Puts the user process into non-recoverable imprecise trapping mode and returns the previous mode. **FP_TRAP_IMP_REC** Puts the user process into recoverable imprecise trapping mode and returns the previous mode. **FP_TRAP_FASTMODE** Puts the user process into the fastest trapping mode available on the hardware platform. **Note:** Some hardware models do not support all modes. If an unsupported mode is requested, the **fp_trap** subroutine returns **FP_TRAP_UNIMPL**.

Return Values

If called with the **FP_TRAP_OFF**, **FP_TRAP_IMP**, **FP_TRAP_IMP_REC**, or **FP_TRAP_SYNC** flag, the **fp_trap** subroutine returns a value indicating which flag was in the previous mode of the process if the hardware

supports the requested mode. If the hardware does not support the requested mode, the **fp_trap** subroutine returns **FP_TRAP_UNIMPL**.

If called with the **FP_TRAP_QUERY** flag, the **fp_trap** subroutine returns a value indicating the current mode of the process, either the **FP_TRAP_OFF**, **FP_TRAP_IMP**, **FP_TRAP_IMP_REC**, or **FP_TRAP_SYNC** flag.

If called with **FP_TRAP_FASTMODE**, the **fp_trap** subroutine sets the fastest mode available and returns the mode selected.

Error Codes

If the **fp_trap** subroutine is called with an invalid parameter, the subroutine returns **FP_TRAP_ERROR**.

If the requested mode is not supported on the hardware platform, the subroutine returns **FP_TRAP_UNIMPL**.

fp_trapstate Subroutine

Purpose

Queries or changes the trapping mode in the Machine Status register (MSR).

Note: This subroutine replaces the **fp_cpusync** subroutine. The **fp_cpusync** subroutine is supported for compatibility, but the **fp_trapstate** subroutine should be used for development.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fptrap.h>
```

```
int fp_trapstate (int)
```

Description

The **fp_trapstate** subroutine is a service routine used to query or set the trapping mode. The trapping mode determines whether floating-point exceptions can generate traps, and can affect execution speed. See **Floating-Point Exceptions Overview** in *General Programming Concepts: Writing and Debugging Programs* for a description of precise and imprecise trapping modes. Floating-point traps can be generated by the hardware only when the processor is in a traps-enabled mode.

The **fp_trapstate** subroutine changes only the trapping mode. It is a service routine for use in developing custom floating-point exception-handling software. If you are using the **fp_enable** or **fp_enable_all** subroutine or the **fp_sh_info** or **fp_sh_set_stat** subroutine, you must use the **fp_trap** subroutine to change the process' trapping mode.

Parameters

Item	Description
flag	Specifies a query of, or change in, the trap mode: **FP_TRAPSTATE_OFF** Sets the trapping mode to Off and returns the previous mode. **FP_TRAPSTATE_QUERY** Returns the current trapping mode without modifying it. **FP_TRAPSTATE_IMP** Puts the process in non-recoverable imprecise trapping mode and returns the previous state. **FP_TRAPSTATE_IMP_REC** Puts the process in recoverable imprecise trapping mode and returns the previous state. **FP_TRAPSTATE_PRECISE** Puts the process in precise trapping mode and returns the previous state. **FP_TRAPSTATE_FASTMODE** Puts the process in the fastest trap-generating mode available on the hardware platform and returns the state selected. **Note:** Some hardware models do not support all modes. If an unsupported mode is requested, the **fp_trapstate** subroutine returns **FP_TRAP_UNIMPL** and the trapping mode is not changed.

Return Values

If called with the **FP_TRAPSTATE_OFF**, **FP_TRAPSTATE_IMP**, **FP_TRAPSTATE_IMP_REC**, or **FP_TRAPSTATE_PRECISE** flag, the **fp_trapstate** subroutine returns a value indicating the previous mode of the process. The value may be **FP_TRAPSTATE_OFF**, **FP_TRAPSTATE_IMP**, **FP_TRAPSTATE_IMP_REC**, or **FP_TRAPSTATE_PRECISE**. If the hardware does not support the requested mode, the **fp_trapstate** subroutine returns **FP_TRAP_UNIMPL**.

If called with the FP_TRAPSTATE_QUERY flag, the **fp_trapstate** subroutine returns a value indicating the current mode of the process. The value may be **FP_TRAPSTATE_OFF**, **FP_TRAPSTATE_IMP**, **FP_TRAPSTATE_IMP_REC**, or **FP_TRAPSTATE_PRECISE**.

If called with the **FP_TRAPSTATE_FASTMODE** flag, the **fp_trapstate** subroutine returns a value indicating which mode was selected. The value may be **FP_TRAPSTATE_OFF**, **FP_TRAPSTATE_IMP**, **FP_TRAPSTATE_IMP_REC**, or **FP_TRAPSTATE_PRECISE**.

fpclassify Macro

Purpose

Classifies real floating type.

Syntax

```
#include <math.h>

int fpclassify(x)
real-floating x;
```

Description

The **fpclassify** macro classifies the *x* parameter as NaN, infinite, normal, subnormal, zero, or into another implementation-defined category. An argument represented in a format wider than its semantic type is converted to its semantic type. Classification is based on the type of the argument.

Parameters

Item	Description
x	Specifies the value to be classified.

Return Values

The **fpclassify** macro returns the value of the number classification macro appropriate to the value of its argument.

fread or fwrite Subroutine

Purpose

Reads and writes binary files.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
size_t fread (Pointer, Size, NumberOfItems, Stream)
void *Pointer;
size_tSize, NumberOfItems;
FILE *Stream;
size_t fwrite (Pointer, Size, NumberOfItems, Stream)
const void *Pointer;
size_t Size, NumberOfItems;
FILE *Stream;
```

Description

The **fread** subroutine copies the number of data items specified by the *NumberOfItems* parameter from the input stream into an array beginning at the location pointed to by the *Pointer* parameter. Each data item has the form ***Pointer*.

The **fread** subroutine stops copying bytes if an end-of-file (EOF) or error condition is encountered while reading from the input specified by the *Stream* parameter, or when the number of data items specified by the *NumberOfItems* parameter have been copied. This subroutine leaves the file pointer of the *Stream* parameter, if defined, pointing to the byte following the last byte read. The **fread** subroutine does not change the contents of the *Stream* parameter.

The `st_atime` field will be marked for update by the first successful run of the **fgetc**, **fgets**, **fgetwc**, **fgetws**, **fread**, **fscanf**, **getc**, **getchar**, **gets**, or **scanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc**or **ungetwc** subroutine.

Note: The **fread** subroutine is a buffered **read** subroutine library call. It reads data in 4KB blocks. For tape block sizes greater than 4KB, use the **open** subroutine and **read** subroutine.

The **fwrite** subroutine writes items from the array pointed to by the *Pointer* parameter to the stream pointed to by the *Stream* parameter. Each item's size is specified by the *Size* parameter. The **fwrite** subroutine writes the number of items specified by the *NumberOfItems* parameter. The file-position

indicator for the stream is advanced by the number of bytes successfully written. If an error occurs, the resulting value of the file-position indicator for the stream is indeterminate.

The **fwrite** subroutine appends items to the output stream from the array pointed to by the *Pointer* parameter. The **fwrite** subroutine appends as many items as specified in the *NumberOfItems* parameter.

The **fwrite** subroutine stops writing bytes if an error condition is encountered on the stream, or when the number of items of data specified by the *NumberOfItems* parameter have been written. The **fwrite** subroutine does not change the contents of the array pointed to by the *Pointer* parameter.

The `st_ctime` and `st_mtime` fields will be marked for update between the successful run of the **fwrite** subroutine and the next completion of a call to the **fflush** or **fclose** subroutine on the same stream, the next call to the **exit** subroutine, or the next call to the **abort** subroutine.

Parameters

Item	Description
Pointer	Points to an array.
Size	Specifies the size of the variable type of the array pointed to by the *Pointer* parameter. The *Size* parameter can be considered the same as a call to **sizeof** subroutine.
NumberOfItems	Specifies the number of items of data.
Stream	Specifies the input or output stream.

Return Values

The **fread** and **fwrite** subroutines return the number of items actually transferred. If the *NumberOfItems* parameter contains a 0, no characters are transferred, and a value of 0 is returned. If the *NumberOfItems* parameter contains a negative number, it is translated to a positive number, since the *NumberOfItems* parameter is of the unsigned type.

Error Codes

If the **fread** subroutine is unsuccessful because the I/O stream is unbuffered or data needs to be read into the I/O stream's buffer, it returns one or more of the following error codes:

Item	Description
EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor specified by the *Stream* parameter, and the process would be delayed in the **fread** operation.
EBADF	Indicates that the file descriptor specified by the *Stream* parameter is not a valid file descriptor open for reading.
EINTR	Indicates that the read operation was terminated due to receipt of a signal, and no data was transferred.

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding **sa_restart**.

Item	Description
EIO	Indicates that the process is a member of a background process group attempting to perform a read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group has no parent process.
ENOMEM	Indicates that insufficient storage space is available.
ENXIO	Indicates that a request was made of a nonexistent device.

If the **fwrite** subroutine is unsuccessful because the I/O stream is unbuffered or the I/O stream's buffer needs to be flushed, it returns one or more of the following error codes:

Item	Description
EAGAIN	Indicates that the **O_NONBLOCK** or O_NDELAY flag is set for the file descriptor specified by the *Stream* parameter, and the process is delayed in the write operation.
EBADF	Indicates that the file descriptor specified by the *Stream* parameter is not a valid file descriptor open for writing.
EFBIG	Indicates that an attempt was made to write a file that exceeds the file size of the process limit or the systemwide maximum file size.
EINTR	Indicates that the write operation was terminated due to the receipt of a signal, and no data was transferred.
EIO	Indicates that the process is a member of a background process group attempting to perform a write to its controlling terminal, the **TOSTOP** signal is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process is orphaned.
ENOSPC	Indicates that there was no free space remaining on the device containing the file.
EPIPE	Indicates that an attempt is made to write to a pipe or first-in-first-out (FIFO) process that is not open for reading by any process. A **SIGPIPE** signal is sent to the process.

The **fwrite** subroutine is also unsuccessful due to the following error conditions:

Item	Description
ENOMEM	Indicates that insufficient storage space is available.
ENXIO	Indicates that a request was made of a nonexistent device, or the request was outside the capabilities of the device.

freehostent Subroutine

Purpose

To free memory allocated by getipnodebyname and getipnodebyaddr.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <netdb.h>
void freehostent (ptr)
struct hostent * ptr;
```

Description

The **freehostent** subroutine frees any dynamic storage pointed to by elements of *ptr*. This includes the **hostent** structure and the data areas pointed to by the h_name, h_addr_list, and h_aliases members of the **hostent** structure.

freelocale Subroutine

Purpose

Frees resources allocated for a locale object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <locale.h>

void freelocale(locobj);
locale_t locobj;
```

Return Value

None

Errors

None

Description

The **freelocale** subroutine releases the resources allocated for a locale object that is returned by a call to the **newlocale** or **duplocale** subroutines.

Any use of a locale object that has been freed results in undefined behavior.

Example

The following example shows a code snippets to free a locale object created by the **newlocale** subroutine:

```
#include <locale.h>

...
/* Every locale object allocated with newlocale() should be
* freed using freelocale():
*/

locale_t loc;
/* Get the locale. */

loc = newlocale (LC_CTYPE_MASK | LC_TIME_MASK, "locname", NULL);
/* ... Use the locale object ... */
...
/* Free the locale object resources. */
freelocale (loc);
```

freelmb Subroutine

Purpose

Returns a block of memory allocated by **alloclmb()** to the system.

Syntax

```
#include <sys/dr.h>
int freelmb(long long laddr
```

Description

The **freelmb()** subroutine returns a block of memory, allocated by **allocmb()**, for general system use.

Parameters

Item	Description
laddr	A previously allocated LMB address.

Execution Environment

This **freelmb()** interface should only be called from the process environment.

Return Values

Item	Description
0	The LMB is successfully freed.

Error Codes

Item	Description
ENOTSUP	LMB allocation not supported on this system.
EINVAL	*laddr* does not describe a previously allocated LMB.
EINVAL	Not in the process environment.

frevoke Subroutine

Purpose

Revokes access to a file by other processes.

Library

Standard C Library (**libc.a**)

Syntax

```
int frevoke ( FileDescriptor)
int FileDescriptor;
```

Description

The **frevoke** subroutine revokes access to a file by other processes.

All accesses to the file are revoked, except through the file descriptor specified by the *FileDescriptor* parameter to the **frevoke** subroutine. Subsequent attempts to access the file, using another file descriptor established before the **frevoke** subroutine was called, fail and cause the process to receive a return value of -1, and the **errno** global variable is set to **EBADF** .

A process can revoke access to a file only if its effective user ID is the same as the file owner ID or if the invoker has root user authority.

Note: The **frevoke** subroutine has no affect on subsequent attempts to open the file. To ensure exclusive access to the file, the caller should change the mode of the file before issuing the **frevoke** subroutine. Currently the **frevoke** subroutine works only on terminal devices.

Parameters

Item	Description
FileDescriptor	A file descriptor returned by a successful **open** subroutine.

Return Values

Upon successful completion, the **frevoke** subroutine returns a value of 0.

If the **frevoke** subroutine fails, it returns a value of -1 and the **errno** global variable is set to indicate the error.

Error Codes

The **frevoke** subroutine fails if the following is true:

Item	Description
EBADF	The *FileDescriptor* value is not the valid file descriptor of a terminal.
EPERM	The effective user ID of the calling process is not the same as the file owner ID.
EINVAL	Revocation of access rights is not implemented for this file.

frexpd32, frexpd64, and frexpd128 Subroutines

Purpose

Extracts the mantissa and exponent from a decimal floating-point number.

Syntax

```
#include <math.h>

_Decimal32 frexpd32 (num, exp)
_Decimal32 num;
int *exp;

_Decimal64 frexpd64 (num, exp)
_Decimal64 num;
int *exp;

_Decimal128 frexpd128 (num, exp)
_Decimal128 num;
int *exp;
```

Description

The **frexpd32**, **frexpd64**, and **frexpd128** subroutines divide a decimal floating-point number into a mantissa and an integral power of 10. The integer exponent is stored in the **int** object pointed to by the **exp** parameter.

Parameters

Item	Description
num	Specifies the decimal floating-point number to be divided into a mantissa and an integral power of 10.
exp	Points to where the integer exponent is stored.

Return Values

For finite arguments, the **frexpd32**, **frexpd64**, and **frexpd128** subroutines return the mantissa value in the **x** parameter. Therefore, the **num** parameter equals the **x** parameter times 10 raised to the power *exp* parameter.

If *num* is NaN, a NaN is returned, and the value of the **exp* is not specified.

If *num* is ±0, ±0 is returned, and the value of the **exp* is 0.

If *num* is ±Inf, *num* is returned, and the value of the **exp* is not specified.

frexpf, frexpl, or frexp Subroutine

Purpose

Extracts the mantissa and exponent from a double precision number.

Syntax

```
#include <math.h>

float frexpf (num, exp)
float num;
int *exp;

long double frexpl (num, exp)
long double num;
int *exp;

double frexp (num, exp)
double num;
int *exp;
```

Description

The **frexpf**, **frexpl**, and **frexp** subroutines break a floating-point number *num* into a normalized fraction and an integral power of 2. The integer exponent is stored in the **int** object pointed to by *exp*.

Parameters

Item	Description
num	Specifies the floating-point number to be broken into a normalized fraction and an integral power of 2.
exp	Points to where the integer exponent is stored.

Return Values

For finite arguments, the **frexpf**, **frexpl**, and **frexp** subroutines return the value *x*, such that *x* has a magnitude in the interval [½ ,1) or 0, and *num* equals *x* times 2 raised to the power *exp*.

If *num* is NaN, a NaN is returned, and the value of **exp* is unspecified.

If *num* is ±0, ±0 is returned, and the value of **exp* is 0.

If *num* is ±Inf, *num* is returned, and the value of **exp* is unspecified.

fscntl Subroutine

Purpose

Controls file system control operations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <j2/j2_cntl.h>
#include <sys/vmount.h>
```

```
int fscntl ( vfs_id,  Command,  Argument,   ArgumentSize)
int vfs_id;
int Command;
char *Argument;
int ArgumentSize;
```

Description

The **fscntl** subroutine performs a variety of file system-specific functions. These functions typically require root user authority.

The Enhanced Journaled File System (JFS2) supports several *Command* values that can be used by applications. Each of these *Command* values requires root authority.

FSCNTL_FREEZE
The file system specified by *vfs_id* is "frozen" for a specified amount of time. The act of freezing a file system produces a nearly consistent on-disk image of the file system, and writes all dirty file system metadata and user data to the disk. In its frozen state, the file system is read-only, and anything that attempts to modify the file system or its contents must wait for the freeze to end. The *Argument* is treated as an integral timeout value in seconds (instead of a pointer). The file system is thawed by FSCNTL_THAW or when the timeout expires. The timeout, which must be a positive value, can be renewed using FSCNTL_REFREEZE. The *ArgumentSize* must be 0.

Note: For all applications using this interface, use FSCNTL_THAW to thaw the file system rather than waiting for the timeout to expire. If the timeout expires, an error log entry is generated as an advisory.

FSCNTL_REFREEZE
The file system specified by *vfs_id*, which must be already frozen, has its timeout value reset. If the command is used on a file system that is not frozen, an error is returned. The *Argument* is treated as an integral timeout value in seconds (instead of a pointer). The file system is thawed by FSCNTL_THAW or when the new timeout expires. The timeout must be a positive value. The *ArgumentSize* must be 0.

FSCNTL_THAW
The file system specified by *vfs_id* is thawed. Modifications to the file system are still allowed after it is thawed, and the file system image might no longer be consistent after the thaw occurs. If the file

system is not frozen at the time of the call, an error is returned. The *Argument* and *ArgumentSize* must both be 0.

The Journaled File System (JFS) supports only internal **fscntl** interfaces. Application programs should not call this function on a JFS file system, because **fscntl** is reserved for system management commands, such as the **chfs** command.

Parameters

Item	Description
vfs_id	Identifies the file system to be acted upon. This information is returned by the **stat** subroutine in the `st_vfs` field of the **stat.h** file.
Command	Identifies the operation to be performed.
Argument	Specifies a pointer to a block of file system specific information that defines how the operation is to be performed.
ArgumentSize	Defines the size of the buffer pointed to by the *Argument* parameter.

Return Values

Upon successful completion, the **fscntl** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fscntl** subroutine fails if any of the following errors are true:

Item	Description
EINVAL	The *vfs_id* parameter does not identify a valid file system.
EINVAL	The *Command* parameter is not recognized by the file system.
EINVAL	The timeout specified to `FSCNTL_FREEZE` or `FSCNTL_REFREEZE` is invalid.
EALREADY	The *Command* parameter was `FSCNTL_FREEZE` and the file system specified was already frozen.
EALREADY	The *Command* parameter was `FSCNTL_REFREEZE` or `FSCNTL_THAW` and the file system specified was not frozen.

fseek, fseeko, fseeko64, rewind, ftell, ftello, ftello64, fgetpos, fgetpos64, fsetpos, or fsetpos64 Subroutine

Purpose

Repositions the file pointer of a stream.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>

int fseek ( Stream,  Offset,  Whence)
FILE *Stream;
```

```
long int Offset;
int Whence;
```

```
void rewind (Stream)
FILE *Stream;
```

```
long int ftell (Stream)
FILE *Stream;
```

```
int fgetpos (Stream, Position)
FILE *Stream;
fpos_t *Position;
```

```
int fsetpos (Stream,  Position)
FILE *Stream;
const fpos_t *Position;
```

```
int fseeko ( Stream,  Offset,  Whence)
FILE *Stream;
off_t Offset;
int Whence;
```

```
int fseeko64 ( Stream,  Offset,  Whence)
FILE *Stream;
off64_t Offset;
int Whence;
```

```
off_t int ftello (Stream)
FILE *Stream;
```

```
off64_t int ftello64 (Stream)
FILE *Stream;
```

```
int fgetpos64 (Stream, Position)
FILE *Stream;
fpos64_t *Position;
```

```
int fsetpos64 (Stream,  Position)
FILE *Stream;
const fpos64_t *Position;
```

Description

The **fseek**, **fseeko** and **fseeko64** subroutines set the position of the next input or output operation on the I/O stream specified by the Stream parameter. The position if the next operation is determined by the *Offset* parameter, which can be either positive or negative.

The **fseek**, **fseeko** and **fseeko64** subroutines set the file pointer associated with the specified *Stream* as follows:

- If the *Whence* parameter is set to the **SEEK_SET** value, the pointer is set to the value of the *Offset* parameter.
- If the *Whence* parameter is set to the **SEEK_CUR** value, the pointer is set to its current location plus the value of the *Offset* parameter.
- If the *Whence* parameter is set to the **SEEK_END** value, the pointer is set to the size of the file plus the value of the *Offset* parameter.

The **fseek**, **fseeko**, and **fseeko64** subroutine are unsuccessful if attempted on a file that has not been opened using the **fopen** subroutine. In particular, the **fseek** subroutine cannot be used on a terminal or on a file opened with the **popen** subroutine. The **fseek** and **fseeko** subroutines will also fail when the resulting offset is larger than can be properly returned.

The **rewind** subroutine is equivalent to calling the **fseek** subroutine using parameter values of (*Stream*,**SEEK_SET,SEEK_SET**), except that the **rewind** subroutine does not return a value. Do not use the **rewind** subroutine in situations where the **fseek** subroutine might fail (for example, when the **fseek** subroutine is used with buffered I/O streams). In this case, use the **fseek** subroutine, so error conditions can be checked.

The **fseek**, **fseeko**, **fseeko64** and **rewind** subroutines undo any effects of the **ungetc** and **ungetwc** subroutines and clear the end-of-file (EOF) indicator on the same stream.

The **fseek**, **fseeko**, and **fseeko64** function allows the file-position indicator to be set beyond the end of existing data in the file. If data is written later at this point, subsequent reads of data in the gap will return bytes of the value 0 until data is actually written into the gap.

A successful calls to the **fsetpos** or **fsetpos64** subroutines clear the **EOF** indicator and undoes any effects of the **ungetc** and **ungetwc** subroutines.

After an **fseek**, **fseeko**, **fseeko64** or a **rewind** subroutine, the next operation on a file opened for update can be either input or output.

ftell, **ftello** and **ftello64** subroutines return the position current value of the file-position indicator for the stream pointed to by the *Stream* parameter. **ftell** and **ftello** will fail if the resulting offset is larger than can be properly returned.

The **fgetpos** and **fgetpos64** subroutines store the current value of the file-position indicator for the stream pointed to by the *Stream* parameter in the object pointed to by the *Position* parameter. The **fsetpos** and **fsetpos64** set the file-position indicator for *Stream* according to the value of the *Position* parameter, which must be the result of a prior call to **fgetpos** or **fgetpos64** subroutine. **fgetpos** and **fsetpos** will fail if the resulting offset is larger than can be properly returned.

Parameters

Item	Description
Stream	Specifies the input/output (I/O) stream.
Offset	Determines the position of the next operation.
Whence	Determines the value for the file pointer associated with the *Stream* parameter.
Position	Specifies the value of the file-position indicator.

Return Values

Upon successful completion, the **fseek**, **fseeko** and **fseeko64** subroutine return a value of 0. Otherwise, it returns a value of -1.

Upon successful completion, the **ftell**, **ftello** and **ftello64** subroutine return the offset of the current byte relative to the beginning of the file associated with the named stream. Otherwise, a **long int** value of -1 is returned and the **errno** global variable is set.

Upon successful completion, the **fgetpos**, **fgetpos64**, **fsetpos** and **fsetpos64** subroutines return a value of 0. Otherwise, a nonzero value is returned and the **errno** global variable is set to the specific error.

Error Codes

If the **fseek**, **fseeko**, **fseeko64**, **ftell**, **ftello**, or **ftello64** subroutines are unsuccessful because the stream is unbuffered or the stream buffer needs to be flushed and the call to the subroutine causes an underlying **lseek** or **write** subroutine to be invoked, it returns one or more of the following error codes:

Item	Description
EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor, delaying the process in the write operation.

Item	Description
EBADF	Indicates that the file descriptor underlying the *Stream* parameter is not open for writing.
EFBIG	Indicates that an attempt has been made to write to a file that exceeds the file-size limit of the process or the maximum file size.
EFBIG	Indicates that the file is a regular file and that an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
EINTR	Indicates that the write operation has been terminated because the process has received a signal, and either no data was transferred, or the implementation does not report partial transfers for this file.
EIO	Indicates that the process is a member of a background process group attempting to perform a **write** subroutine to its controlling terminal, the **TOSTOP** flag is set, the process is not ignoring or blocking the **SIGTTOU** signal, and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.
ENOSPC	Indicates that no remaining free space exists on the device containing the file.
EPIPE	Indicates that an attempt has been made to write to a pipe or FIFO that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.
EINVAL	Indicates that the *Whence* parameter is not valid. The resulting file-position indicator will be set to a negative value. The **EINVAL** error code does not apply to the **ftell** and **rewind** subroutines.
ESPIPE	Indicates that the file descriptor underlying the *Stream* parameter is associated with a pipe, FIFO, or socket.
EOVERFLOW	Indicates that for *fseek*, the resulting file offset would be a value that cannot be represented correctly in an object of type *long*.
EOVERFLOW	Indicates that for *fseeko*, the resulting file offset would be a value that cannot be represented correctly in an object of type *off_t*.
ENXIO	Indicates that a request was made of a non-existent device, or the request was outside the capabilities of the device.

The **fgetpos** and **fsetpos** subroutines are unsuccessful due to the following conditions:

Item	Description
EINVAL	Indicates that either the *Stream* or the *Position* parameter is not valid. The **EINVAL** error code does not apply to the **fgetpos** subroutine.
EBADF	Indicates that the file descriptor underlying the *Stream* parameter is not open for writing.
ESPIPE	Indicates that the file descriptor underlying the *Stream* parameter is associated with a pipe, FIFO, or socket.

The **fseek**, **fseeko**, **ftell**, **ftello**, **fgetpos**, and **fsetpos** subroutines are unsuccessful under the following condition:

Item	Description
EOVERFLOW	The resulting could not be returned properly.

fsync or fsync_range Subroutine

Purpose

Writes changes in a file to permanent storage.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int fsync ( FileDescriptor)
int FileDescriptor;

int fsync_range (FileDescriptor, how, start, length)
int FileDescriptor;
int how;
off_t start;
off_t length;
```

Description

The **fsync** subroutine causes all modified data in the open file specified by the *FileDescriptor* parameter to be saved to permanent storage. On return from the **fsync** subroutine, all updates have been saved on permanent storage.

The **fsync_range** subroutine causes all modified data in the specified range of the open file specified by the *FileDescriptor* parameter to be saved to permanent storage. On return from the **fsync_range** subroutine, all updates in the specified range have been saved on permanent storage.

This paragraph refers to deprecated function available only in the JFS file system. Data written to a file that a process has opened for deferred update (with the **O_DEFER** flag) is not written to permanent storage until another process issues an **fsync_range** or **fsync** call against this file or runs a synchronous **write** subroutine (with the **O_SYNC** flag) on this file. See the **fcntl.h** file and the **open** subroutine for descriptions of the **O_DEFER** and **O_SYNC** flags respectively.

Note: The file identified by the *FileDescriptor* parameter must be open for writing when the **fsync** subroutine is issued or the call is unsuccessful. This restriction was not enforced in BSD systems. The **fsync_range** subroutine does not require write access.

Parameters

Item	Description
FileDescriptor	A valid, open file descriptor.

Item	Description
how	Specify the handling characteristics of the operation. **O_SYNC** The modified data in the range specified by the <*start, length*> parameters is written to storage. If any metadata is modified then all modified user data is written to storage. Any metadata changes and the file attributes including timestamps are also written to storage. **O_DSYNC** The modified data in the range specified by the <*start, length*> parameters is written to storage. If there is modified metadata for the file then the metadata is also written if it is required to read the data. Otherwise, no metadata updates occur. **O_NOCACHE** The modified data is written as with the **O_DSYNC** parameter. The full pages in the range specified by the <*start, length*> parameters are removed from the memory cache. The pages are removed from the cache even if they are not modified. The operation also works on files that are open only for reading.
start	Starting file offset.
length	Length, or zero for all cache data.

Return Values

Upon successful completion, the **fsync** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Upon successful completion, the **fsync_range** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **fsync** or **fsync_range** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EIO	An I/O error occurred while reading from or writing to the file system.
EBADF	The *FileDescriptor* parameter is not a valid file descriptor open for writing.
EINVAL	The file is not a regular file.
EINTR	The subroutine was interrupted by a signal.

ftok Subroutine

Purpose

Generates a standard interprocess communication key.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
```

key_t ftok (*Path*, *ID*)
char **Path*;
int *ID*;

Description

Attention: If the *Path* parameter of the **ftok** subroutine names a file that has been removed while keys still refer to it, the **ftok** subroutine returns an error. If that file is then re-created, the **ftok** subroutine will probably return a key different from the original one.

Attention: Each installation should define standards for forming keys. If standards are not adhered to, unrelated processes may interfere with each other's operation.

Attention: The **ftok** subroutine does not guarantee unique key generation. However, the occurrence of key duplication is very rare and mostly for across file systems.

The **ftok** subroutine returns a key, based on the *Path* and *ID* parameters, to be used to obtain interprocess communication identifiers. The **ftok** subroutine returns the same key for linked files if called with the same *ID* parameter. Different keys are returned for the same file if different *ID* parameters are used.

All interprocess communication facilities require you to supply a key to the **msgget**, **semget**, and **shmget** subroutines in order to obtain interprocess communication identifiers. The **ftok** subroutine provides one method for creating keys, but other methods are possible. For example, you can use the project ID as the most significant byte of the key, and use the remaining portion as a sequence number.

Parameters

Item	Description
Path	Specifies the path name of an existing file that is accessible to the process.
ID	Specifies a character that uniquely identifies a project.

Return Values

When successful, the **ftok** subroutine returns a key that can be passed to the **msgget**, **semget**, or **shmget** subroutine.

Error Codes

The **ftok** subroutine returns the value **(key_t)-1** if one or more of the following are true:

- The file named by the *Path* parameter does not exist.
- The file named by the *Path* parameter is not accessible to the process.
- The *ID* parameter has a value of 0.

ftw or ftw64 Subroutine

Purpose

Walks a file tree.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ftw.h>

int ftw ( Path,  Function,  Depth)
char *Path;
int (*Function(const char*, const struct stat*, int);
int Depth;

int ftw64 ( Path,  Function,  Depth)
char *Path;
int (*Function(const char*, const struct stat64*, int);
int Depth;
```

Description

The **ftw** and **ftw64** subroutines recursively searches the directory hierarchy that descends from the directory specified by the *Path* parameter.

For each file in the hierarchy, the **ftw** and **ftw64** subroutines call the function specified by the *Function* parameter. **ftw** passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, and an integer. **ftw64** passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a **stat64** structure containing information about the file, and an integer.

The integer passed to the *Function* parameter identifies the file type with one of the following values:

Item	Description
FTW_F	Regular file
FTW_D	Directory
FTW_DNR	Directory that cannot be read
FTW_SL	Symbolic Link
FTW_NS	File for which the **stat** structure could not be executed successfully

If the integer is **FTW-DNR**, the files and subdirectories contained in that directory are not processed.

If the integer is **FTW-NS**, the **stat** structure contents are meaningless. An example of a file that causes **FTW-NS** to be passed to the *Function* parameter is a file in a directory for which you have read permission but not execute (search) permission.

The **ftw** and **ftw64** subroutines finish processing a directory before processing any of its files or subdirectories.

The **ftw** and **ftw64** subroutines continue the search until the directory hierarchy specified by the *Path* parameter is completed, an invocation of the function specified by the *Function* parameter returns a nonzero value, or an error is detected within the **ftw** and **ftw64** subroutines, such as an I/O error.

The **ftw** and **ftw64** subroutines traverse symbolic links encountered in the resolution of the *Path* parameter, including the final component. Symbolic links encountered while walking the directory tree rooted at the *Path* parameter are not traversed.

The **ftw** and **ftw64** subroutines use one file descriptor for each level in the tree. The *Depth* parameter specifies the maximum number of file descriptors to be used. In general, the **ftw** and **ftw64** subroutines runs faster if the value of the *Depth* parameter is at least as large as the number of levels in the tree. However, the value of the *Depth* parameter must not be greater than the number of file descriptors currently available for use. If the value of the *Depth* parameter is 0 or a negative number, the effect is the same as if it were 1.

Because the **ftw** and **ftw64** subroutines are recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **ftw** and **ftw64** subroutines use the **malloc** subroutine to allocate dynamic storage during its operation. If the **ftw** and **ftw64** subroutined is terminated prior to its completion, such as by the **longjmp** subroutine being executed by the function specified by the *Function* parameter or by an interrupt routine, the **ftw** and **ftw64** subroutines cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the *Function* parameter return a nonzero value the next time it is called.

Parameters

Item	Description
Path	Specifies the directory hierarchy to be searched.
Function	Specifies the file type.
Depth	Specifies the maximum number of file descriptors to be used. *Depth* cannot be greater than OPEN_MAX which is described in the sys/limits.h header file.

Return Values

If the tree is exhausted, the **ftw** and **ftw64** subroutines returns a value of 0. If the subroutine pointed to by **fn** returns a nonzero value, **ftw** and **ftw64** subroutines stops its tree traversal and returns whatever value was returned by the subroutine pointed to by **fn**. If the **ftw** and **ftw64** subroutines detects an error, it returns a -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **ftw** or **ftw64** subroutines detect an error, a value of -1 is returned and the **errno** global variable is set to indicate the error.

The **ftw** and **ftw64** subroutine are unsuccessful if:

Item	Description
EACCES	Search permission is denied for any component of the *Path* parameter or read permission is denied for *Path*.
ENAMETOOLONG	The length of the path exceeds **PATH_MAX** while **_POSIX_NO_TRUNC** is in effect.
ENOENT	The *Path* parameter points to the name of a file that does not exist or points to an empty string.
ENOTDIR	A component of the *Path* parameter is not a directory.

The **ftw** subroutine is unsuccessful if:

Item	Description
EOVERFLOW	A file in *Path* is of a size larger than 2 Gigabytes.

fwide Subroutine

Purpose

Set stream orientation.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>
```

```
int fwid (FILE * stream, int mode),
```

Description

The **fwide** function determines the orientation of the stream pointed to by stream. If mode is greater than zero, the function first attempts to make the stream wide-oriented. If mode is less than zero, the function first attempts to make the stream byte-oriented. Otherwise, mode is zero and the function does not alter the orientation of the stream.

If the orientation of the stream has already been determined, **fwide** does not change it.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, then call **fwide**, then check errno and if it is non-zero, assume an error has occurred.

A call to **fwide** with mode set to zero can be used to determine the current orientation of a stream.

Return Values

The **fwide** function returns a value greater than zero if, after the call, the stream has wide-orientation, a value less than zero if the stream has byte-orientation, or zero if the stream has no orientation.

Errors

The **fwide** function may fail if:

Item	Description
EBADF	The stream argument is not a valid stream.

fwprintf, wprintf, swprintf Subroutines

Purpose

Print formatted wide-character output.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>
```

```
int fwprintf ( FILE * stream, const wchar_t * format, . . .)
int wprintf (const wchar_t * format, . .)
int swprintf (wchar_t *s, size_t n, const wchar_t * format, . . .)
```

Description

The **fwprintf** function places output on the named output stream. The **wprintf** function places output on the standard output stream **stdout**. The **swprintf** function places output followed by the null wide-

character in consecutive wide-characters starting at ***s**; no more than **n** wide-characters are written, including a terminating null wide-character, which is always added (unless **n** is zero).

Each of these functions converts, formats and prints its arguments under control of the **format** wide-character string. The **format** is composed of zero or more directives: **ordinary wide-characters**, which are simply copied to the output stream and **conversion specifications** , each of which results in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the **format.** If the **format** is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

EX Conversions can be applied to the **nth** argument after the **format** in the argument list, rather than to the next unused argument. In this case, the conversion wide-character % (see below) is replaced by the sequence **%n$**, where n is a decimal integer in the range [1, {NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format wide-character strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

In format wide-character strings containing the **%n$** form of conversion specifications, numbered arguments in the argument list can be referenced from the format wide-character string as many times as required.

In format wide-character strings containing the % form of conversion specifications, each argument in the argument list is used exactly once.

All forms of the **fwprintf** functions allow for the insertion of a language-dependent radix character in the output string, output as a wide-character value. The radix character is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

EX Each conversion specification is introduced by the % wide-character or by the wide-character sequence %n$,after which the following appear in sequence:

- Zero or more **flags** (in any order), which modify the meaning of the conversion specification.
- An optional minimum **field width**. If the converted value has fewer wide-characters than the field width, it will be padded with spaces by default on the left; it will be padded on the right, if the left-adjustment flag (-), described below, is given to the field width. The field width takes the form of an asterisk (*), described below, or a decimal integer.
- An optional **precision** that gives the minimum number of digits to appear for the d, i, o, u, **x** and **X** conversions; the number of digits to appear after the radix character for the e, E and f conversions; the maximum number of significant digits for the g and G conversions; or the maximum number of wide-characters to be printed from a string in s conversions. The precision takes the form of a period (.) followed either by an asterisk (*), described below, or an optional decimal digit string, where a null digit string is treated as 0. If a precision appears with any other conversion wide-character, the behaviour is undefined.
- An optional **l** (lowercase *L*), **L**, **h**, **H**, **D** or **DD** specifies one of the following conversions:
  - An optional **l** specifying that a following **c** conversion wide-character applies to a **wint_t** argument.
  - An optional **l** specifying that a following **s** conversion wide-character applies to a **wchar_t** argument.
  - An optional **l** specifying that a following **d**, **i**, **o**, **u**, **x** or **X** conversion wide-character applies to a type **long int** or **unsigned long int** argument.
  - An optional **l** specifying that a following **n** conversion wide-character applies to a pointer to a type **long int** argument.
  - An optional **L** specifying that a following **e**, **E**, **f**, **g** or **G** conversion wide-character applies to a type **long double** argument.
  - An optional **h** specifying that a following **d**, **i**, **o**, **u**, **x** or **X** conversion wide-character applies to a type **short int** or type **unsigned short int** argument (the argument that will be promoted according to the integral promotions, and its value will be converted to type **short int** or **unsigned short int** before printing).
  - An optional **h** specifying that a following **n** conversion wide-character applies to a pointer to a type **short int** argument.

- An optional **H** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion wide-character applies to a **_Decimal32** parameter.
- An optional **D** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion wide-character applies to a **_Decimal64** parameter.
- An optional **DD** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion wide-character applies to a **_Decimal128** parameter.

If an **l**, **L** , **h**, **H**, **D**, or **DD** appears with any other conversion wide-character, the behavior is undefined.

- A **conversion wide-character** that indicates the type of conversion to be applied.

A field width, or precision, or both, may be indicated by an asterisk (*). In this case an argument of type int supplies the field width or precision. Arguments specifying field width, or precision, or both must appear in that order before the argument, if any, to be converted. A negative field width is taken as a - flag followed by a positive field width. A negative precision is taken as if EX the precision were omitted. In format wide-character strings containing the %n$ form of a conversion specification, a field width or precision may be indicated by the sequence *m$, where m is a decimal integer in the range [1, {NL_ARGMAX}] giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
wprintf(L"%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The **format** can contain either numbered argument specifications (that is, **%n$** and ***m$),** or unnumbered argument specifications (that is, % and *), but normally not both. The only exception to this is that %% can be mixed with the **%n$** form. The results of mixing numbered and unnumbered argument specifications in a **format** wide-character string are undefined. When numbered argument specifications are used, specifying the Nth argument requires that all the leading arguments, from the first to the (N-1)th, are specified in the format wide-character string.

The flag wide-characters and their meanings are:

Item	Description
'	The integer portion of the result of a decimal conversion (%i, %d, %u, %f, %g or %G) will be formatted with thousands' grouping wide-characters. For other conversions the behaviour is undefined. The non-monetary grouping wide-character is used.
-	The result of the conversion will be left-justified within the field. The conversion will be right-justified if this flag is not specified.
+	The result of a signed conversion will always begin with a sign (+ or -). The conversion will begin with a sign only when a negative value is converted if this flag is not specified.
space	If the first wide-character of a signed conversion is not a sign or if a signed conversion results in no wide-characters, a space will be prefixed to the result. This means that if the space and + flags both appear, the space flag will be ignored.
#	This flag specifies that the value is to be converted to an alternative form. For o conversion, it increases the precision (if necessary) to force the first digit of the result to be 0. For x or X conversions, a non-zero result will have 0x (or 0X) prefixed to it. For e, E, f, g or G conversions, the result will always contain a radix character, even if no digits follow it. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For g and G conversions, trailing zeros will **not** be removed from the result as they normally are. For other conversions, the behavior is undefined.
0	For d, i, o, u, x, X, e, E, f, g and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and - flags both appear, the 0 flag will be ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag will be ignored. If the 0 and ' flags both appear, the grouping wide-characters are inserted before zero padding. For other conversions, the behavior is undefined.

The conversion wide-characters and their meanings are:

Item	Description
d,i	The **int** argument is converted to a signed decimal in the style [-] **dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
o	The **unsigned int** argument is converted to unsigned octal format in the style **dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
u	The **unsigned int** argument is converted to unsigned decimal format in the style **dddd**. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
x	The **unsigned int** argument is converted to unsigned hexadecimal format in the style **dddd**; the letters abcdef are used. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting 0 with an explicit precision of 0 is no wide-characters.
X	Behaves the same as the x conversion wide-character except that letters ABCDEF are used instead of abcdef.
f	The **double** argument is converted to decimal notation in the style [-] **ddd.ddd**, where the number of digits after the radix character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly 0 and no # flag is present, no radix character appears. If a radix character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits. The **fwprintf** family of functions may make available wide-character string representations for infinity and NaN.
e, E	The **double** argument is converted in the style [-] **d.ddde +/- dd**, where there is one digit before the radix character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is 0 and no # flag is present, no radix character appears. The value is rounded to the appropriate number of digits. The E conversion wide-character will produce a number with E instead of e introducing the exponent. The exponent always contains at least two digits. If the value is 0, the exponent is 0. The **fwprintf** family of functions may make available wide-character string representations for infinity and NaN.
g, G	The **double** argument is converted in the style f or e (or in the style E in the case of a G conversion wide-character), with the precision specifying the number of significant digits. If an explicit precision is 0, it is taken as 1. The style used depends on the value converted; style e (or E) will be used only if the exponent resulting from such a conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a radix character appears only if it is followed by a digit. The **fwprintf** family of functions may make available wide-character string representations for infinity and NaN.
c	If no l (ell) qualifier is present, the **int** argument is converted to a wide-character as if by calling the **btowc** function and the resulting wide-character is written. Otherwise the **wint_t** argument is converted to **wchar_t,** and written.

Item	Description
s	If no l (ell) qualifier is present, the argument must be a pointer to a character array containing a character sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the **mbrtowc** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first character is converted, and written up to (but not including) the terminating null wide-character. If the precision is specified, no more than that many wide-characters are written. If the precision is not specified or is greater than the size of the array, the array must contain a null wide-character. If an l (ell) qualifier is present, the argument must be a pointer to an array of type **wchar_t.** Wide characters from the array are written up to (but not including) a terminating null wide-character. If no precision is specified or is greater than the size of the array, the array must contain a null wide-character. If a precision is specified, no more than that many wide-characters are written.
p	The argument must be a pointer to void. The value of the pointer is converted to a sequence of printable wide-characters, in an implementation-dependent manner. The argument must be a pointer to an integer into which is written the number of wide-characters written to the output so far by this call to one of the **fwprintf** functions. No argument is converted.
C	Same as lc.
S	Same as ls.
%	Output a % wide-character; no argument is converted. The entire conversion specification must be %%.

If a conversion specification does not match one of the above forms, the behavior is undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by **fwprintf** and **wprintf** are printed as if **fputwc** had been called.

The **st_ctime** and **st_mtime** fields of the file will be marked for update between the call to a successful execution of **fwprintf** or **wprintf** and the next successful completion of a call to **fflush** or **fclose** on the same stream or a call to exit or abort.

Return Values

Upon successful completion, these functions return the number of wide-characters transmitted excluding the terminating null wide-character in the case of **swprintf** or a negative value if an output error was encountered.

Error Codes

For the conditions under which **fwprintf** and **wprintf** will fail and may fail, refer to **fputwc**. In addition, all forms of **fwprintf** may fail if:

Item	Description
EILSEQ	A wide-character code that does not correspond to a valid character has been detected
EINVAL	There are insufficient arguments.
	In addition, **wprintf** and **fwprintf** may fail if:
ENOMEM	Insufficient storage space is available.

The **swprintf** will fail if:

Item	Description
EOVERFLOW	The value of *n* is greater than {INT_MAX} or the number of bytes needed to hold the output excluding the terminating null is greater than {INT_MAX}.

Examples

To print the language-independent date and time format, the following statement could be used:

```
wprintf (format, weekday, month, day, hour, min);
```

For American usage, format could be a pointer to the wide-character string:

```
L"%s, %s %d, %d:%.2d\n"
```

producing the message:

```
Sunday, July 3, 10:02
```

whereas for German usage, format could be a pointer to the wide-character string:

```
L"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

producing the message:

```
Sonntag, 3. July, 10:02
```

fwscanf, wscanf, swscanf Subroutines

Purpose

Convert formatted wide-character input.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>
```

```
int fwscanf (FILE * stream, const wchar_t * format, ...);
int wscanf (const wchar_t * format, ...);
int swscanf (const wchar_t * s, const wchar_t * format, ...);
```

Description

The **fwscanf** function reads from the named input stream. The **wscanf** function reads from the standard input stream stdin. The **swscanf** function reads from the wide-character string s. Each function reads wide-characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control wide-character string format described below, and a set of pointer arguments indicating where the converted input should be stored. The result is undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the **nth** argument after the **format** in the argument list, rather than to the next unused argument. In this case, the conversion wide-character % (see below) is replaced by the sequence **%n$,** where **n** is a decimal integer in the range [1, {NL_ARGMAX}]. This feature provides for

the definition of format wide-character strings that select arguments in an order appropriate to specific languages. In format wide-character strings containing the **%n$** form of conversion specifications, it is unspecified whether numbered arguments in the argument list can be referenced from the format wide-character string more than once.

The format can contain either form of a conversion specification, that is, % or **%n$,** but the two forms cannot normally be mixed within a single format wide-character string. The only exception to this is that %% or %* can be mixed with the **%n$** form.

The **fwscanf** function in all its forms allows for detection of a language-dependent radix character in the input string, encoded as a wide-character value. The radix character is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period (.).

The format is a wide-character string composed of zero or more directives. Each directive is composed of one of the following: one or more white-space wide-characters (space, tab, newline, vertical-tab or form-feed characters); an ordinary wide-character (neither % nor a white-space character); or a conversion specification. Each conversion specification is introduced by a % or the sequence **%n$** after which the following appear in sequence:

- An optional assignment-suppressing character *.
- An optional non-zero decimal integer that specifies the maximum field width.
- An optional size modifier **h**, **H**, **l** (lowercase *L*), **L**, **D**, or **DD** indicating the size of the receiving object.
  - Must precede the **c**, **s** and **[** conversion wide-characters with the **l** (lowercase *L*) if the corresponding argument is a pointer to **wchar_t**.
  - Must precede the **d**, **i** and **n** conversion wide-characters with the **h** if the corresponding argument is a pointer to **short int** or with the **l** (lowercase *L*) if it is a pointer to **long int**.
  - Must precede the **o**, **u** and **x** conversion wide-characters with the **h** if the corresponding argument is a pointer to **unsigned short int** or with **l** (lowercase *L*) if it is a pointer to **unsigned long int**.
  - Must precede the **e**, **f** and **g** conversion wide-characters with **l** (lowercase *L*) if the corresponding argument is a pointer to **double** or with the **L** if it is a pointer to long double.
  - Must precede the **e**, **f** and **g** conversion wide-characters with the **H** if the corresponding argument is a pointer to**_Decimal32**.
  - Must precede the **e**, **f** and **g** conversion wide-characters with the **D** if the corresponding argument is a pointer to**_Decimal64**.
  - Must precede the **e**, **f** and **g** conversion wide-characters with the **DD** if the corresponding argument is a pointer to**_Decimal128**.

  If an **l** (lowercase *L*), **L**, **h**, **H**, **D**, or **DD** appears with any other conversion wide-character, the behavior is undefined.
- A conversion wide-character that specifies the type of conversion to be applied. The valid conversion wide-characters are described below.

The **fwscanf** functions execute each directive of the format in turn. If a directive fails, as detailed below, the function returns. Failures are described as input failures (due to the unavailability of input bytes) or matching failures (due to inappropriate input).

A directive composed of one or more white-space wide-characters is executed by reading input until no more valid input can be read, or up to the first wide-character which is not a white-space wide-character, which remains unread.

A directive that is an ordinary wide-character is executed as follows. The next wide-character is read from the input and compared with the wide-character that comprises the directive; if the comparison shows that they are not equivalent, the directive fails, and the differing and subsequent wide-characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each conversion wide-character. A conversion specification is executed in the following steps:

Input white-space wide-characters (as specified by **iswspace**) are skipped, unless the conversion specification includes a [, c or n conversion character.

An item is read from the input, unless the conversion specification includes an n conversion wide-character. An input item is defined as the longest sequence of input wide-characters, not exceeding any specified field width, which is an initial subsequence of a matching sequence. The first wide-character, if any, after the input item remains unread. If the length of the input item is 0, the execution of the conversion specification fails; this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % conversion wide-character, the input item (or, in the case of a %n conversion specification, the count of input wide-characters) is converted to a type appropriate to the conversion wide-character. If the input item is not a matching sequence, the execution of the conversion specification fails; this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the **format** argument that has not already received a conversion result if the conversion specification is introduced by %, or in the nth argument if introduced by the wide-character sequence %n$. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion wide-characters are valid:

Item	Description
d	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of **wcstol** with the value 10 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **int**.
i	Matches an optionally signed integer, whose format is the same as expected for the subject sequence of **wcstol** with 0 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **int**.
o	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of **wcstoul** with the value 8 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **unsigned int**.
u	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of **wcstoul** with the value 10 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **unsigned int**.
x	Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of **wcstoul** with the value 16 for the base argument. In the absence of a size modifier, the corresponding argument must be a pointer to **unsigned int**.
e, f, g	Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of **wcstod** . In the absence of a size modifier, the corresponding argument must be a pointer to float. If the **fwprintf** family of functions generates character string representations for infinity and NaN (a 7858 symbolic entity encoded in floating-point format) to support the ANSI/IEEE Std 754:1985 standard, the **fwscanf**5 family of functions will recognise them as input.
s	Matches a sequence of non white-space wide-characters. If no l (ell) qualifier is present, characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically. Otherwise, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide-character, which will be added automatically.

Item	Description
[	Matches a non-empty sequence of wide-characters from a set of expected wide-characters (the **scanset**). If no l (ell) qualifier is present, wide-characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence and the terminating null character, which will be added automatically. If an l (ell) qualifier is present, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence and the terminating null wide-character, which will be added automatically The conversion specification includes all subsequent wide characters in the **format** string up to and including the matching right square bracket (]). The wide-characters between the square brackets (the **scanlist**) comprise the scanset, unless the wide-character after the left square bracket is a circumflex (^), in which case the scanset contains all wide-characters that do not appear in the scanlist between the circumflex and the right square bracket. If the conversion specification begins with [ ] or [^], the right square bracket is included in the scanlist and the next right square bracket is the matching right square bracket that ends the conversion specification; otherwise the first right square bracket is the one that ends the conversion specification. If a - is in the scanlist and is not the first wide-character, nor the second where the first wide-character is a ^;, nor the last wide-character, the behavior is implementation-dependent.
c	Matches a sequence of wide-characters of the number specified by the field width (1 if no field width is present in the conversion specification). If no l (ell) qualifier is present, wide-characters from the input field are converted as if by repeated calls to the **wcrtomb** function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide-character is converted. The corresponding argument must be a pointer to a character array large enough to accept the sequence. No null character is added. Otherwise, the corresponding argument must be a pointer to an array of **wchar_t** large enough to accept the sequence. No null wide-character is added.
p	Matches an implementation-dependent set of sequences, which must be the same as the set of sequences that is produced by the %p conversion of the corresponding **fwprintf** functions. The corresponding argument must be a pointer to a pointer to void. The interpretation of the input item is implementation-dependent. If the input item is a value converted earlier during the same program execution, the pointer that results will compare equal to that value; otherwise the behavior of the %p conversion is undefined.
n	No input is consumed. The corresponding argument must be a pointer to the integer into which is to be written the number of wide-characters read from the input so far by this call to the **fwscanf** functions. Execution of a %n conversion specification does not increment the assignment count returned at the completion of execution of the function.
C	Same as lc.
S	Same as ls.
%	Matches a single %; no conversion or assignment occurs. The complete conversion specification must be %%.

If a conversion specification is invalid, the behavior is undefined.

The conversion characters E, G and X are also valid and behave the same as, respectively, e, g and x.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide-characters matching the current conversion specification (except for %n) have been read (other than leading white-space, where permitted), execution of the current conversion specification terminates with an input failure. Otherwise, unless execution of the current conversion specification is terminated

with a matching failure, execution of the following conversion specification (if any) is terminated with an input failure.

Reaching the end of the string in **swscanf** is equivalent to encountering end-of-file for **fwscanf**.

If conversion terminates on a conflicting input, the offending input is left unread in the input. Any trailing white space (including newline) is left unread unless matched by a conversion specification. The success of literal matches and suppressed assignments is only directly determinable via the %n conversion specification.

The **fwscanf** and **wscanf** functions may mark the **st_atime** field of the file associated with stream for update. The **st_atime** field will be marked for update by the first successful execution of **fgetc**, **fgetwc**, **fgets**, **fgetws**, **fread**, **getc**, **getwc**, **getchar**, **getwchar**, **gets**, **fscanf** or **fwscanf** using stream that returns data not supplied by a prior call to **ungetc**.

In format strings containing the % form of conversion specifications, each argument in the argument list is used exactly once.

Return Values

Upon successful completion, these functions return the number of successfully matched and assigned input items; this number can be 0 in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF is returned. If a read error occurs the error indicator for the stream is set, EOF is returned, and errno is set to indicate the error.

Error Codes

For the conditions under which the **fwscanf** functions will fail and may fail, refer to **fgetwc**. In addition, **fwscanf** may fail if:

Item	Description
EILSEQ	Input byte sequence does not form a valid character.
EINVAL	There are insufficient arguments.

Examples

The call:

```
int i, n; float x; char name[50];
n = wscanf(L"%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 Hamster
```

will assign to n the value 3, to i the value 25, to x the value 5.432, and name will contain the string Hamster.

The call:

```
int i; float x; char name[50];
(void) wscanf(L"%2d%f%*d %[0123456789]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign 56 to **i**, 789.0 to x, skip 0123, and place the string 56\0 in **name**. The next call to **getchar** will return the character a.

g

The following Base Operating System (BOS) runtime services begin with the letter *g*.

gai_strerror Subroutine

Purpose

Facilitates consistent error information from EAI_* values returned by the getaddrinfo subroutine.

Library

Library (**libc.a**)

Syntax

```
#include <sys/socket.h>
#include <netdb.h>
char *
gai_strerror (ecode)
int ecode;
int
gai_strerror_r (ecode, buf, buflen)
int ecode;
char *buf;
int buflen;
```

Description

For multithreaded environments, the second version should be used. In **gai_strerror_r**, *buf* is a pointer to a data area to be filled in. *buflen* is the length (in bytes) available in *buf*.

It is the caller's responsibility to insure that *buf* is sufficiently large to store the requested information, including a trailing null character. It is the responsibility of the function to insure that no more than *buflen* bytes are written into *buf*.

Return Values

If successful, a pointer to a string containing an error message appropriate for the EAI_* errors is returned. If ecode is not one of the EAI_* values, a pointer to a string indicating an unknown error is returned.

gamma Subroutine

Purpose

Computes the natural logarithm of the gamma function.

Libraries

The **gamma**: IEEE Math Library (**libm.a**) or System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
```

```
extern int signgam;
```

```
double gamma (x)
double x;
```

Description

The **gamma** subroutine computes the logarithm of the gamma function.

The sign of gamma(*x*) is returned in the external integer **signgam**.

Note: Compile any routine that uses subroutines from the **libm.a** with the **-lm** flag. To compile the **lgamma.c** file, enter:

```
cc lgamma.c -lm
```

Parameters

Item	Description
x	Specifies the value to be computed.

garbagedlines Subroutine

Purpose

Discards and replaces a number of lines in a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
garbagedlines(Window,  BegLine, NumLines)
WINDOW * Window;
int BegLine,  NumLines;
```

Description

The **garbagedlines** subroutine discards and replaces lines in a window. The *Begline* parameter specifies the beginning line number and the *Numlines* parameter specifies the number of lines to discard. Curses discards and replaces the specified lines before adding more data.

Uses this subroutine for applications that need to redraw a line that is garbled. Lines may become garbled as the result of noisy communication lines. Instead of refreshing the entire display, use the **garbagedlines** subroutine to refresh a portion of the display and to avoid even more communication noise.

Parameters

Item	Description
Window	Points to a window.
BegLine	Identifies the beginning line in a range of lines to discard.

Item	Description
NumLines	Specifies the total number of lines in a range of lines to discard and replace.

Examples

To discard and replace 5 lines in the `mywin` window starting with line `10`, use:

```
WINDOW *mywin; garbagedlines(mywin, 10, 5);
```

gencore or coredump Subroutine

Purpose

Creates a core file without terminating the process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <core.h>

int gencore (coredumpinfop)
struct coredumpinfo *coredumpinfop;

int coredump (coredumpinfop)
struct coredumpinfo *coredumpinfop;
```

Description

The **gencore** and **coredump** subroutines create a core file of a process without terminating it. The core file contains the snapshot of the process at the time the call is made and can be used with the **dbx** command for debugging purposes.

If any thread of the process is in a system call when its snapshot core file is generated, the register information returned may not be reliable (except for the stack pointer). To save all user register contents when a system call is made so that they are available to the **gencore** and **coredump** subroutines, the application should be built using the `"-bM:UR"` flags.

If any thread of the process is sleeping inside the kernel or stopped (possibly for job control), the caller of the **gencore** and **coredump** subroutines will also be blocked until the thread becomes runnable again. Thus, these subroutines may take a long time to complete depending upon the target process state.

The **coredump** subroutine always generates a core file for the process from which it is called. This subroutine has been replaced by the **gencore** subroutine and is being provided for compatibility reasons only.

The **gencore** subroutine creates a core file for the process whose process ID is specified in the *pid* field of the **coredumpinfo** structure. For security measures, the user ID (uid) and group ID (gid) of the core file are set to the uid and gid of the process.

Both these subroutines return success even if the core file cannot be created completely because of filesystem space constraints. When using the **dbx** command with an incomplete core file, **dbx** may warn that the core file is truncated.

In the **"Change / Show Characteristics of Operating System"** smitty screen, there are two options regarding the creation of the core file. The core file will always be created in the default core format and will ignore the value specified in the **"Use pre-430 style CORE dump"** option. However, the value

specified for the **"Enable full CORE dump"** option will be considered when creating the core file. Resource limits of the target process for **file** and **coredump** will be enforced.

The **coredumpinfo** structure contains the following fields:

Member Type	Member Name	Description
unsigned int	*length*	Length of the core file name.
char *	*name*	Name of the core file.
pid_t	*pid*	ID of the process to be coredumped.
int	*flags*	Flags-version flag. Set this to **GENCORE_VERSION_1**.

Note: The *pid* and *flags* fields are required for the **gencore** subroutine, but are ignored for the **coredump** subroutine

Parameters

Item	Description
coredumpinfop	Specifies the address of the **coredumpinfo** structure that provides the file name to save the core snapshot and its length. For the **gencore** subroutine, it also provides the process id of the process whose core is to be dumped and a flag which includes version flag bits. The version flag value must be set to **GENCORE_VERSION_1**.

Return Values

Upon successful completion, the **gencore** and **coredump** subroutines return a 0. If unsuccessful, a -1 is returned, and the **errno** global variable is set to indicate the error

Error Codes

Item	Description
EACCES	Search permission is denied on a component of the path prefix, the file exists and permissions specified by the mode are denied, or the file does not exist and write permission is denied for the parent directory of the file to be created.
ENOENT	The *name* field in the *coredumpinfo* parameter points to an empty string.
EINTR	The subroutine was interrupted by a signal before it could complete.
ENAMETOOLONG	The value of the *length* field in the **coredumpinfop** structure or the length of the absolute path of the specified core file name is greater than **MAXPATHLEN** (as defined in the **sys/param.h** file).
EINVAL	The value of the *length* field in the **coredumpinfop** structure is 0.
EAGAIN	The target process is already in the middle of another **gencore** or **coredump** subroutine.
ENOMEM	Unable to allocate memory resources to complete the subroutine.

In addition to the above, the following **errno** values can be set when the **gencore** subroutine is unsuccessful:

Item	Description
EPERM	The real or effective user ID of the calling process does not match the real or effective user ID of target process or the calling process does not have root user authority.
ESRCH	There is no process whose ID matches the value specified in the *pid* field of the *coredumpinfop* parameter or the process is exiting.
EINVAL	The *flags* field in the *coredumpinfop* parameter is not set to a valid version value.

genpagvalue Subroutine

Purpose

Sets the current process credentials.

Library

Security Library (libc.a)

Syntax

```
#include <pag.h>
int  genpagvalue(pag_name, pag_value,pag_flags);
char *       pag_name;
uint64_t *   pag_value;
int           pag_flags;
```

Description

The **genpagvalue** subroutine generates a new PAG value for a given PAG name. For this function to succeed, the PAG name must be registered with the operating system before calling the **genpagvalue** subroutine. The **genpagvalue** subroutine is limited to maintaining information about the last generated PAG number and accordingly generating a new number. This service can optionally store the PAG value in the process's **cred** structure. It does not monitor the PAG values stored in the **cred** structure by other means.

The PAG value returned is of size 64 bits. The number of significant bits is determined by the requested PAG type. 32-bit PAGs have 32 significant bits. 64-bit PAGs have 62 significant bits.

A process must have root authority to invoke this function for 32-bit PAG types. Any process may invoke this function for 64-bit PAG types.

The *pag_flags* parameter with the value **PAG_SET_VALUE** causes the generated value to be atomically stored in the process's credentials. The *pag_flags* parameter with both the **PAG_SET_VALUE** and **PAG_COPY_CRED** values set causes the current process's credentials to be duplicated before the generated value is stored.

Parameters

Item	Description
pag_name	The name parameter is a 1 to 4 character, NULL terminated name for the PAG type. Typical values include `afs`, `dfs`, `pki` and `krb5`.
pag_value	This pointer points to a buffer where the OS will return the newly generated PAG value.

Item	Description
pag_flags	These flags control the behavior of the **getpagvalue** subroutine. This must be set to 0 or one or more of the values **PAG_SET_VALUE** or **PAG_COPY_CRED**.

Return Values

A value of 0 is returned upon successful completion. If the **genpagvalue** subroutine fails a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **genpagvalue** subroutine fails if one or more of the following are true:

Item	Description
EINVAL	The PAG value cannot be generated because the named PAG type does not exist as part of the table.
EPERM	The process does not have the correct authority to use the service.

Other errors might be set by subroutines invoked by the **genpagvalue** subroutine.

get_ipc_info Subroutine

Purpose

Get IPC information for a requested workload partition.

Syntax

```
#include <sys/ipc_info.h>

int get_ipc_info(cid, cmd, version, buffer, size)
cid_t cid;
int cmd;
int version;
char * buffer;
int * size;
```

Description

The **get_ipc_info** subroutine returns IPC information for the associated workload partition ID and copies it to the address specified for the *buffer* parameter. If *cid* parameter is zero, then the IPC information of the workload partition that is associated to the current process is returned. Based on the command specified for *cmd* that is requested, an array of corresponding structures will be copied to the address starting at the address specified for *buffer*. The number of array structures depends on the number of IPC objects of the requested type that are present.

The value specified for the *cid* parameter is not used as input to the **GET_IPCINFO_SHM_ALL**, **GET_IPCINFO_MSG_ALL**, and **GET_IPCINFO_SEM_ALL** commands. These commands are useful from the global workload partition to return IPC information for all workload partitions on the system.

If the value for the *size* parameter on input is smaller than the data to be returned, then **ENOSPC** is returned and the value for the *size* parameter is set to the actual size needed.

Parameters

Item	Description
cid	Specifies the workload partition ID.

Item	Description
cmd	Specifies which request command to perform. See **cmd types** for a list of possible commands.
version	Specifies which version of the request structure to return. Valid versions are specified in the **sys/ipc_info.h** header file.
buffer	Specifies the starting address for the requested IPC structures.
size	Specifies the maximum number of bytes to return.

Cmd types

The *cmd* parameter is supplied on input and describes the type of IPC information to return. The following *cmd* types are supported:

Item	Description
GET_IPCINFO_SHM	Returns System V shared memory structures *ipcinfo_shm_t* for the requested workload partition.
GET_IPCINFO_MSG	Returns System V message queue structures *ipcinfo_msg_t* for the requested workload partition.
GET_IPCINFO_SEM	Returns System V semaphore structures *ipcinfo_sem_t* for the requested workload partition.
GET_IPCINFO_RTSHM	Returns POSIX real-time shared memory structures *ipcinfo_rtshm_t* for the requested workload partition.
GET_IPCINFO_RTMSG	Returns POSIX real-time message queue structures *ipcinfo_rtmq_t* for the requested workload partition.
GET_IPCINFO_RTSEM	Returns POSIX real-time semaphore structures *ipcinfo_rtsem_t* for the requested workload partition.
GET_IPCINFO_SHM_ALL	Returns all System V shared memory structures *ipcinfo_shm_t* that are accessible by the current process.
GET_IPCINFO_MSG_ALL	Returns all System V message queue structures *ipcinfo_msg_t* that are accessible by the current process.
GET_IPCINFO_SEM_ALL	Returns all System V semaphore structures *ipcinfo_sem_t* that are accessible by the current process.

Execution Environment

Process environment only.

Return Values

Item	Description
0	The command completed successfully.
EPERM	Error indicating the current process does not have permission to retrieve workload partition information for the WPAR ID specified for the *cid* parameter.
EINVAL	Invalid value specified for the *cmd*, *version*, or *cid* parameters.
EFAULT	Error during the **copyout** to user space.
ENOSPC	Size for the *buffer* parameter that is indicated by the *size* parameter is smaller than the data to be returned.

get_malloc_log Subroutine

Purpose

Retrieves information about the malloc subsystem.

Syntax

```
#include <malloc.h>
size_t get_malloc_log (addr, buf, bufsize)
void *addr;
void *buf;
size_t bufsize;
```

Description

The **get_malloc_log** subroutine retrieves a record of currently active malloc allocations. These records are stored as an array of **malloc_log** structures, which are copied from the process heap into the buffer specified by the *buf* parameter. No more than *bufsize* bytes are copied into the buffer. Only records corresponding to the heap of which *addr* is a member are copied, unless *addr* is NULL, in which case records from all heaps are copied. The *addr* parameter must be either a pointer to space allocated previously by the malloc subsystem or NULL.

Parameters

Item	Description
addr	Pointer to a space allocated by the malloc subsystem.
buf	Specifies into which buffer the **malloc_log** structures are stored.
bufsize	Specifies the number of bytes that can be copied into the buffer.

Return Values

The **get_malloc_log** subroutine returns the number of bytes actually transferred into the *bufsize* parameter. If Malloc Log is not enabled, 0 is returned. If *addr* is not a pointer allocated by the malloc subsystem, 0 is returned and the **errno** global variable is set to **EINVAL**.

get_malloc_log_live Subroutine

Purpose

Provides information about the malloc subsystem.

Syntax

```
#include <malloc.h>
struct malloc_log* get_malloc_log_live (addr)
void *addr;
```

Description

The **get_malloc_log_live** subroutine provides access to a record of currently active malloc allocations. The information is stored as an array of **malloc_log** structures, which are located in the process heap. This data is volatile and subject to update. The *addr* parameter must be either a pointer to space allocated previously by the malloc subsystem or NULL.

Parameters

Item	Description
addr	Pointer to space allocated previously by the malloc subsystem

Return Values

The **get_malloc_log_live** subroutine returns a pointer to the process heap at which the records of current malloc allocations are stored. If the *addr* parameter is NULL, a pointer to the beginning of the array is returned. If *addr* is a pointer to space allocated previously by the malloc subsystem, the pointer returned corresponds to records of the same heap as *addr*. If Malloc Log is not enabled, NULL is returned. If *addr* is not a pointer allocated by the malloc subsystem, NULL is returned and the **errno** global variable is set to **EINVAL**.

get_speed, set_speed, or reset_speed Subroutines

Purpose

Set and get the terminal baud rate.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/str_tty.h>
```

```
int get_speed (FileDescriptor)
int FileDescriptor;
```

```
int set_speed (FileDescriptor, Speed)
int FileDescriptor;
int Speed;
```

```
int reset_speed (FileDescriptor)
int FileDescriptor;
```

Description

The baud rate functions **set_speed** subroutine and **get_speed** subroutine are provided top allow the user applications to program any value of the baud rate that is supported by the asynchronous adapter, but that cannot be expressed using the termios subroutines **cfsetospeed**, **cfsetispeed**, **cfgetospeed**, and **cfsgetispeed**. Those subroutines are indeed limited to the set values {BO, B50, ..., B38400} described in <**termios.h**>.

Interaction with the termios Baud flags:

If the terminal's device driver supports these subroutines, it has two interfaces for baud rate manipulation.

Operation for Baud Rate:

normal mode: This is the default mode, in which a termios supported speed is in use.

speed-extended mode: This mode is entered either by calling **set_speed** subroutine a non-termios supported speed at the configuration of the line.

In this mode, all the calls to **tcgetattr** subroutine or **TCGETS ioctl** subroutine will have B50 in the returned termios structure.

If **tcsetatt** subroutine or **TCSETS**, **TCSETAF**, or **TCSETAW ioctl** subroutines is called and attempt to set B50, the actual baud rate is not changed. If is attempts to set any other termios-supported speed, the driver will switch back to the normal mode and the requested baud rate is set. Calling **reset_speed** subroutine is another way to switch back to the normal mode.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.
Speed	The integer value of the requested speed.

Return Values

Upon successful completion, **set_speed** and **reset_speed** return a value of 0, and **get_speed** returns a positive integer specifying the current speed of the line. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EINVAL	The *FileDescriptor* parameter does not specify a valid file descriptor for a **tty** the recognizes the **set_speed**, **get_speed** and **reset_speed** subroutines, or the *Speed* parameter of **set_speed** is not supported by the terminal.

Plus all the **errno** codes that may be set in case of failure in an **ioctl** subroutine issued to a streams based **tty**.

getargs Subroutine

Purpose

Gets arguments of a process.

Library

Standard C library (**libc.a)**

Syntax

```
#include <procinfo.h>
#include <sys/types.h>

int getargs (processBuffer, bufferLen, argsBuffer, argsLen)
struct procsinfo  *processBuffer
or struct procentry64  *processBuffer;
int  bufferLen;
char  *argsBuffer;
int  argsLen;
```

Description

The **getargs** subroutine returns a list of parameters that were passed to a command when it was started. Only one process can be examined per call to **getargs**.

The **getargs** subroutine uses the pi_pid field of *processBuffer* to determine which process to look for. *bufferLen* should be set to the size of **struct procsinfo** or **struct procentry64**. Parameters are returned in *argsBuffer*, which should be allocated by the caller. The size of this array must be given in *argsLen*.

On return, *argsBuffer* consists of a succession of strings, each terminated with a null character (ascii `\0'). Hence, two consecutive NULLs indicate the end of the list.

Note: The arguments may be changed asynchronously by the process, but results are not guaranteed to be consistent.

Parameters

processBuffer
Specifies the address of a **procsinfo** or **procentry64** structure, whose pi_pid field should contain the pid of the process that is to be looked for.

bufferLen
Specifies the size of a single **procsinfo** or **procentry64** structure.

argsBuffer
Specifies the address of an array of characters to be filled with a series of strings representing the parameters that are needed. An extra NULL character marks the end of the list. This array must be allocated by the caller.

argsLen
Specifies the size of the *argsBuffer* array. No more than *argsLen* characters are returned.

Return Values

If successful, the **getargs** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getargs** subroutine does not succeed if the following are true:

Item	Description
ESRCH	The specified process does not exist.
EFAULT	The copy operation to the buffer was not successful or the *processBuffer* or *argsBuffer* parameters are invalid.
EINVAL	The *bufferLen* parameter does not contain the size of a single **procsinfo** or **procentry64** structure.
ENOMEM	There is no memory available in the address space.

getaudithostattr, IDtohost, hosttoID, nexthost or putaudithostattr Subroutine

Purpose

Accesses the host information in the audit host database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int  getaudithostattr (Hostname, Attribute, Value, Type)
char *Hostname;
char *Attribute;
void *Value;
int Type;

char *IDtohost (ID);
char *ID;

char *hosttoID (Hostname, Count);
char *Hostname;
int Count;

char *nexthost (void);

int putaudithostattr (Hostname, Attribute, Value, Type);
char *Hostname;
char *Attribute;
void *Value;
int Type;
```

Description

These subroutines access the audit host information.

The **getaudithostattr** subroutine reads a specified attribute from the host database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly the **putaudithostattr** subroutine writes a specified attribute into the host database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putaudithostattr** must be explicitly committed by calling the **putaudithostattr** subroutine with a Type of **SEC_COMMIT**. Until all the data is committed, only these subroutines within the process return written data.

New entries in the host database must first be created by invoking **putaudithostattr** with the **SEC_NEW** type.

The **IDtohost** subroutine converts an 8 byte host identifier into a hostname.

The **hosttoID** subroutine converts a hostname to a pointer to an array of valid 8 byte host identifiers. A pointer to the array of identifiers is returned on success. A **NULL** pointer is returned on failure. The number of known host identifiers is returned in ***Count**.

The **nexthost** subroutine returns a pointer to the name of the next host in the audit host database.

Parameters

Item	Description
Attribute	Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file: **S_AUD_CPUID** Host identifier list. The attribute type is **SEC_LIST**.
Count	Specifies the number of 8 byte host identifier entries that are available in the *IDarray* parameter or that have been returned in the *IDarray* parameter.
Hostname	Specifies the name of the host for the operation.

Item	Description
ID	An 8 byte host identifier.
IDarray	Specifies a pointer to an array of 1 or more 8 byte host identifiers.
Type	Specifies the type of attribute expected. Valid types are defined in **usersec.h**. The only valid Type value is **SEC_LIST**.
Value	The return value for read operations and the new value for write operations.

Return Values

On successful completion, the **getaudithostattr**, **IDtohost**, **hosttoID**, **nexthost**, or **putaudithostattr** subroutine returns 0. If unsuccessful, the subroutine returns non-zero.

Error Codes

The **getaudithostattr**, **IDtohost**, **hosttoID**, **nexthost**, or **putaudithostattr** subroutine fails if the following is true:

Item	Description
EINVAL	If invalid attribute *Name* or if *Count* is equal to zero for the **hosttoID** subroutine.
ENOENT	If there is no matching *Hostname* entry in the database.

getauthattr Subroutine

Purpose

Queries the authorizations that are defined in the authorization database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getauthattr(Auth, Attribute, Value, Type)
    char *Auth;
    char *Attribute;
    void *Value;
    int Type;
```

Description

The **getauthattr** subroutine reads a specified attribute from the authorization database. The **getauthattr** subroutine can retrieve authorization definitions from both the user-defined authorization database and the system-defined authorization table. For attributes of the **SEC_CHAR** and **SEC_LIST** types, the **getauthattr** subroutine returns the value in allocated memory. The caller needs to free this memory.

Parameters

Item	Description
Auth	The authorization name. This parameter must be specified unless the *Type* parameter is **SEC_COMMIT**.
Attribute	Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file: **S_AUTHORIZATIONS** A list of all available authorizations on the system. This attribute is read-only and is only available to the **getauthattr** subroutine when **ALL** is specified for the *Auth* parameter. The attribute type is **SEC_LIST**. **S_AUTH_CHILDREN** A list of all authorizations that exist in the authorization hierarchy below the authorization specified by the *Auth* parameter. This attribute is read-only and is available only to the **getauthattr** subroutine. The attribute type is **SEC_LIST**. **S_DFLTMSG** Specifies the default authorization description to use if message catalogs are not in use. The attribute type is **SEC_CHAR**. **S_ID** Specifies a unique integer that is used to identify the authorization. The attribute type is **SEC_INT**. **Note:** Do not modify this value after it is set initially when the authorization is created. Modifying the value might compromise the security of the system. **S_MSGCAT** Specifies the message catalog file name that contains the description of the authorization. The attribute type is **SEC_CHAR**. **S_MSGSET** Specifies the message set that contains the description of the authorization in the file that the **S_MSGCAT** attribute specifies. The attribute type is **SEC_INT**. **S_MSGNUMBER** Specifies the message number for the description of the authorization in the file that the **S_MSGCAT** attribute specifies and the message set that the **S_MSGSET** attribute specifies. The attribute type is **SEC_INT**. **S_ROLES** A list of roles using this authorization. This attribute is read-only. The attribute type is **SEC_LIST**.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Item	Description
Type	Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include: **SEC_INT** The format of the attribute is an integer. The user should supply a pointer to a defined integer variable. **SEC_CHAR** The format of the attribute is a null-terminated character string. The user should supply a pointer to a defined character pointer variable. The value is returned as allocated memory. The caller needs to free this memory. **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters. The user should supply a pointer to a defined character pointer variable. The value is returned as allocated memory. The caller needs to free this memory.

Security

Files Accessed:

File	Mode
/etc/security/ authorizations	rw

Return Values

If successful, the **getauthattr** subroutine returns 0. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **getauthattr** subroutine fails, one of the following **errno** values can be set:

Item	Description
EINVAL	The *Auth* parameter is **NULL** or one of the reserved authorization names (**default, ALLOW_OWNER, ALLOW_GROUP, ALLOW_ALL**).
EINVAL	The *Attribute* or *Type* parameter is **NULL** or does not contain one of the defined values.
EINVAL	The *Auth* parameter is **ALL** and the *Attribute* parameter is not **S_AUTHORIZATIONS**.
EINVAL	The *Value* parameter does not point to a valid buffer for this type of attribute.
ENOATTR	The *Attribute* parameter is **S_AUTHORIZATIONS**, but the *Auth* parameter is not **ALL**.
ENOATTR	The attribute specified in the *Attribute* parameter is valid but no value is defined for the authorization.
ENOENT	The authorization specified in the *Auth* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

getauthattrs Subroutine

Purpose

Retrieves multiple authorization attributes from the authorization database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getauthattrs(Auth, Attributes, Count)
    char *Auth;
    dbattr_t *Attributes;
    int Count;
```

Description

The **getauthattrs** subroutine reads one or more attributes from the authorization database. The **getauthattrs** subroutine can retrieve authorization definitions from both the user-defined authorization database and the system-defined authorization table.

The *Attributes* array contains information about each attribute that is to be read. Each element in the *Attributes* array must be examined upon a successful call to the **getauthattrs** subroutine, to determine whether the *Attributes* array was successfully retrieved. The attributes of the **SEC_CHAR** or **SEC_LIST** type will have their values returned to allocated memory. The caller need to free this memory. The **dbattr_t data** structure contains the following fields:

Item	Description
attr_name	The name of the target authorization attribute.
attr_idx	This attribute is used internally by the **getauthattrs** subroutine.
attr_type	The type of a target attribute.
attr _flag	The result of the request to read the target attribute. On successful completion, a value of zero is returned. Otherwise, a value of nonzero is returned.
attr_un	A union that contains the returned values for the requested query.
attr_domain	The **getauthattrs** subroutine ignores any input to this field. If this field is set to null, the subroutine sets this field to the name of the domain where the authorization is found.

The following valid authorization attributes for the **getauthattrs** subroutine are defined in the **usersec.h** file:

Name	Description	Type
S_AUTHORIZATIONS	A list of all available authorizations on the system. It is valid only when the *Auth* parameter is set to the **ALL** variable.	**SEC_LIST**
S_AUTH_CHILDREN	A list of all authorizations that exist in the authorization hierarchy under the authorization that is specified by the *Auth* parameter.	**SEC_LIST**

Name	Description	Type
S_DFLTMSG	The default authorization description that is used when catalogs are not in use.	**SEC_CHAR**
S_ID	A unique integer that is used to identify the authorization.	**SEC_INT**
S_MSGCAT	The message catalog name that contains the authorization description.	**SEC_CHAR**
S_MSGSET	The message catalog set number of the authorization description.	**SEC_INT**
S_MSGNUMBER	The message number of the authorization description.	**SEC_INT**
S_ROLES	A list of roles that contain the authorization in their authorization set.	**SEC_LIST**

The following union members correspond to the definitions of the **attr_char**, **attr_int**, **attr_long** and **attr_llong** macros in the **usersec.h** file:

Item	Description
au_char	Attributes of the **SEC_CHAR** and **SEC_LIST** types store a pointer to the returned value in this member when the attributes are successfully retrieved. The caller is responsible for freeing this memory.
au_int	The storage location for attributes of the **SEC_INT** type.
au_long	The storage location for attributes of the **SEC_LONG** type.
au_llong	The storage location for attributes of the **SEC_LLONG** type.

If **ALL** is specified for the *Auth* parameter, the only valid attribute that can be displayed in the *Attribute* array is the **S_AUTHORIZATIONS** attribute. Specifying any other attribute with an authorization name of **ALL** causes the **getauthattrs** subroutine to fail.

Parameters

Item	Description
Auth	Specifies the authorization name for the *Attributes* array to read.
Attributes	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of authorization attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the *Attributes* array.

Security

Files Accessed:

File	Mode
/etc/security/ authorizations	r

Return Values

If the authorization that is specified by the *Auth* parameter exists in the authorization database, the **getauthattrs** subroutine returns the value of zero. On successful completion, the **attr_flag** attribute of each element in the *Attributes* array must be examined to determine whether it was successfully retrieved. If the specified authorization does not exist, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **getauthattrs** subroutine returns -1, one of the following **errno** values is set:

Item	Description
EINVAL	The *Auth* parameter is **NULL, default, ALLOW_OWNER, ALLOW_GROUP**, or **ALLOW_ALL**.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Attributes* array is **NULL** and the *Count* parameter is greater than zero.
EINVAL	The *Auth* parameter is **ALL** but the Attributes entry contains an attribute other than **S_AUTHORIZATIONS**.
ENOENT	The authorization specified in the *Auth* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	Operation is not permitted.
EACCES	Access permission is denied for the data request.

If the **getauthattrs** subroutine fails to query an attribute, one of the following errors is returned to the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The invoker does not have access to the attribute specified in the **attr_name** field.
EINVAL	The **attr_name** field in the *Attributes* entry is not a recognized authorization attribute.
EINVAL	The **attr_type** field in the *Attributes* entry contains a type that is not valid.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer.
ENOATTR	The **attr_name** field in the *Attributes* entry specifies a valid attribute, but no value is defined for this authorization.

getauthdb or getauthdb_r Subroutine

Purpose

Finds the current administrative domain.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getauthdb (Value)
authdb_t *Value;

int getauthdb_r (Value)
authdb_t *Value;
```

Description

The **getauthdb** and **getauthdb_r** subroutines return the value of the current authentication domain in the *Value* parameter. The **getauthdb** subroutine returns the value of the current process-wide authentication domain. The **getauthdb_r** subroutine returns the authentication domain for the current thread if one has been set. The subroutines return -1 if no administrative domain has been set.

Parameters

Item	Description
Value	A pointer to a variable of type **authdb_t**. The **authdb_t** type is a 16-character array that contains the name of a loadable authentication module.

Return Values

Item	Description
1	The value returned is from the process-wide data.
0	The value returned is from the thread-specific data. An authentication database module has been specified by an earlier call to the **setauthdb** subroutine. The name of the current database module has been copied to the *Value* parameter.
-1	The subroutine failed. An authentication database module has not been specified by an earlier call to the **setauthdb** subroutine.

getbegyx, getmaxyx, getparyx, or getyx Subroutine

Purpose

Gets the cursor and window coordinates.

Library

Curses Library (**libcurses.a**)

Syntax

```
include <curses.h>
```

```
void getbegyx(WINDOW  *win,
int  y,
int  x);
```

```
void getmaxyx(WINDOW *win,
int y,
int x);
```

```
void getparyx(WINDOW *win,
int y,
int x);
```

```
void getyx(WINDOW *win,
int y,
int x);
```

Description

The **getbegyx** macro stores the absolute screen coordinates of the specified window's origin in *y* and *x*.

The **getmaxyx** macro stores the number of rows of the specified window in *y* and *x* and stores the window's number of columns in *x*.

The **getparyx** macro, if the specified window is a subwindow, stores in *y* and *x* the coordinates of the window's origin relative to its parent window. Otherwise, -1 is stored in *y* and *x*.

The **getyx** macro stores the cursor position of the specified window in *y* and *x*.

Parameters

Item	Description
**win*	Identifies the window to get the coordinates from.
Y	Returns the row coordinate.
X	Returns the column coordinate.

Examples

For the **getbegyx** subroutine:

To obtain the beginning coordinates for the `my_win` window and store in integers *y* and *x*, use:

```
WINDOW *my_win;
int y, x;
getbegyx(my_win, y, x);
```

For the **getmaxyx** subroutine:

To obtain the size of the `my_win` window, use:

```
WINDOW *my_win;

int y,x;
getmaxyx(my_win, y, x);
```

Integers *y* and *x* will contain the size of the window.

getc, getchar, fgetc, or getw Subroutine

Purpose

Gets a character or word from an input stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

int getc (*Stream***)**
FILE ****Stream*;**

```
int fgetc (Stream)
FILE *Stream;
```

```
int getchar (void)
```

```
int getw (Stream)
FILE *Stream;
```

Description

The **getc** macro returns the next byte as an **unsigned char** data type converted to an **int** data type from the input specified by the *Stream* parameter and moves the file pointer, if defined, ahead one byte in the *Stream* parameter. The **getc** macro cannot be used where a subroutine is necessary; for example, a subroutine pointer cannot point to it.

Because it is implemented as a macro, the **getc** macro does not work correctly with a *Stream* parameter value that has side effects. In particular, the following does not work:

```
getc(*f++)
```

In such cases, use the **fgetc** subroutine.

The **fgetc** subroutine performs the same function as the **getc** macro, but **fgetc** is a true subroutine, not a macro. The **fgetc** subroutine runs more slowly than **getc** but takes less disk space.

The **getchar** macro returns the next byte from **stdin** (the standard input stream). The **getchar** macro is equivalent to **getc(stdin)**.

The first successful run of the **fgetc**, **fgets**, **fgetwc**, **fgetws**, **fread**, **fscanf**, **getc**, **getchar**, **gets** or **scanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc** or **ungetwc** subroutine marks the `st_atime` field for update.

The **getc** and **getchar** macros have also been implemented as subroutines for ANSI compatibility. To access the subroutines instead of the macros, insert **#undef getc** or **#undef getchar** at the beginning of the source file.

The **getw** subroutine returns the next word (**int**) from the input specified by the *Stream* parameter and increments the associated file pointer, if defined, to point to the next word. The size of a word varies from one machine architecture to another. The **getw** subroutine returns the constant **EOF** at the end of the file or when an error occurs. Since **EOF** is a valid integer value, the **feof** and **ferror** subroutines should be used to check the success of **getw**. The **getw** subroutine assumes no special alignment in the file.

Because of additional differences in word length and byte ordering from one machine architecture to another, files written using the **putw** subroutine are machine-dependent and may not be readable using the **getw** macro on a different type of processor.

Parameters

Item	Description
Stream	Points to the file structure of an open file.

Return Values

Upon successful completion, the **getc**, **fgetc**, **getchar**, and **getw** subroutines return the next byte or **int** data type from the input stream pointed by the *Stream* parameter. If the stream is at the end of the file, an end-of-file indicator is set for the stream and the integer constant **EOF** is returned. If a read error occurs, the **errno** global variable is set to reflect the error, and a value of **EOF** is returned. The **ferror** and **feof** subroutines should be used to distinguish between the end of the file and an error condition.

Error Codes

If the stream specified by the *Stream* parameter is unbuffered or data needs to be read into the stream's buffer, the **getc**, **getchar**, **fgetc**, or **getw** subroutine is unsuccessful under the following error conditions:

Item	Description
EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the stream specified by the *Stream* parameter. The process would be delayed in the **fgetc** subroutine operation.
EBADF	Indicates that the file descriptor underlying the stream specified by the *Stream* parameter is not a valid file descriptor opened for reading.
EFBIG	Indicates that an attempt was made to read a file that exceeds the process' file-size limit or the maximum file size. See the **ulimit** subroutine.
EINTR	Indicates that the read operation was terminated due to the receipt of a signal, and either no data was transferred, or the implementation does not report partial transfer for this file. **Note:** Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding **sa_restart**.
EIO	Indicates that a physical error has occurred, or the process is in a background process group attempting to perform a **read** subroutine call from its controlling terminal, and either the process is ignoring (or blocking) the **SIGTTIN** signal or the process group is orphaned.
EPIPE	Indicates that an attempt is made to read from a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.
EOVERFLOW	Indicates that the file is a regular file and an attempt was made to read at or beyond the offset maximum associated with the corresponding stream.

The **getc**, **getchar**, **fgetc**, or **getw** subroutine is also unsuccessful under the following error conditions:

Item	Description
ENOMEM	Indicates insufficient storage space is available.
ENXIO	Indicates either a request was made of a nonexistent device or the request was outside the capabilities of the device.

getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked Subroutines

Purpose

stdio with explicit client locking.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int getc_unlocked (FILE * stream);
int getchar_unlocked (void);
int putc_unlocked (int c, FILE * stream);
int putchar_unlocked (int c);
```

Description

Versions of the functions **getc**, **getchar**, **putc**, and **putchar** respectively named **getc_unlocked**, **getchar_unlocked**, **putc_unlocked**, and **putchar_unlocked** are provided which are functionally identical to the original versions with the exception that they are not required to be implemented in a thread-safe manner. They may only safely be used within a scope protected by **flockfile** (or **ftrylockfile**) and **funlockfile**. These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call of the **flockfile** or **ftrylockfile** functions.

Return Values

See **getc**, **getchar**, **putc**, and **putchar**.

getch, mvgetch, mvwgetch, or wgetch Subroutine

Purpose

Gets a single-byte character from the terminal.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int getch(void)

int mvgetch(int y,
int x);
```

```
int mvwgetch(WINDOW *win,
int y,
int x);

int wgetch(WINDOW *win);
```

Description

The **getch**, **wgetch**, **mvgetch**, and **mvwgetch** subroutines read a single-byte character from the terminal associated with the current or specified window. The results are unspecified if the input is not a single-byte character. If the **keypad** subroutine is enabled, these subroutines respond to the corresponding KEY_ value defined in <curses.h>.

Processing of terminal input is subject to the general rules described in Section 3.5 on page 34.

If echoing is enabled, then the character is echoed as though it were provided as an input argument to the **addch** subroutine, except for the following characters:

```
<backspace>,
```

```
<left-arrow> and
```

```
the current erase character:
```

The input is interpreted as specified in Section 3.4.3 on page 31 and then the character at the resulting cursor position is deleted as though the **delch** subroutine was called, except that if the cursor was originally in the first column of the line, then the user is alerted as though the **beep** subroutine was called.

The user is alerted as though the **beep** subroutine was called. Information concerning the function keys is not returned to the caller.

Function Keys

If the current or specified window is not a pad, and it has been moved or modified since the last refresh operation, then it will be refreshed before another character is read.

The Importance of Terminal Modes

The output of the **getch** subroutines is, in part, determined by the mode of the terminal. The following describes the action of the **getch** subroutines in each type of terminal mode:

Mode	Action of getch Subroutines
NODELAY	Returns a value of ERR if there is no input waiting.
DELAY	Halts execution until the system passes text through the program. If CBREAK mode is also set, the program stops after receiving one character. If **NOCBREAK** mode is set, the **getch** subroutine stops reading after the first new line character.
HALF-DELAY	Halts execution until a character is typed or a specified time out is reached. If **echo** is set, the character is also echoed to the window.

Note: When using the **getch** subroutines do not set both the **NOCBREAK** mode and the **ECHO** mode at the same time. This can cause undesirable results depending on the state of the tty driver when each character is typed.

Getting Function Keys

If your program enables the keyboard with the **keypad** subroutine, and the user presses a function key, the token for that function key is returned instead of raw characters. The possible function keys are defined in the **/usr/include/curses.h** file. Each **#define** macro begins with a **KEY_** prefix.

If a character is received that could be the beginning of a function key (such as an Escape character) curses sets a timer. If the remainder of the sequence is not received before the timer expires, the

character is passed through. Otherwise, the function key's value is returned. For this reason, after a user presses the Esc key there is a delay before the escape is returned to the program. Programmers should not use the Esc key for a single character routine.

Within the **getch** subroutine, a structure of type timeval, defined in the **/usr/include/sys/time.h** file, indicates the maximum number of microseconds to wait for the key response to complete.

The **ESCDELAY** environment variable sets the length of time to wait before timing out and treating the ESC keystroke as the ESC character rather than combining it with other characters in the buffer to create a key sequence. The **ESCDELAY** environment variable is measured in fifths of a millisecond. If **ESCDELAY** is 0, the system immediately composes the **ESCAPE** response without waiting for more information from the buffer. The user may choose any value between 0 and 99,999, inclusive. The default setting for the **ESCDELAY** environment variable is 500 (one tenth of a second).

Programs that do not want the **getch** subroutines to set a timer can call the **notimeout** subroutine. If notimeout is set to TRUE, curses does not distinguish between function keys and characters when retrieving data.

The **getch** subroutines might not be able to return all function keys because they are not defined in the **terminfo** database or because the terminal does not transmit a unique code when the key is pressed. The following function keys may be returned by the **getch** subroutines:

Item	Description
KEY_MIN	Minimum curses key.
KEY_BREAK	Break key (unreliable).
KEY_DOWN	Down Arrow key.
KEY_UP	Up Arrow key.
KEY_LEFT	Left Arrow key.
KEY_RIGHT	Right Arrow key.
KEY_HOME	Home key.
KEY_BACKSPACE	Backspace.
KEY_F(*n*)	Function key F*n*, where *n* is an integer from 0 to 64.
KEY_DL	Delete line.
KEY_IL	Insert line.
KEY_DC	Delete character.
KEY_IC	Insert character or enter insert mode.
KEY_EIC	Exit insert character mode.
KEY_CLEAR	Clear screen.
KEY_EOS	Clear to end of screen.
KEY_EOL	Clear to end of line.
KEY_SF	Scroll 1 line forward.
KEY_SR	Scroll 1 line backwards (reverse).
KEY_NPAGE	Next page.
KEY_PPAGE	Previous page.
KEY_STAB	Set tab.
KEY_CTAB	Clear tab.
KEY_CATAB	Clear all tabs.

Item	Description
KEY_ENTER	Enter or send (unreliable).
KEY_SRESET	Soft (partial) reset (unreliable).
KEY_RESET	Reset or hard reset (unreliable).
KEY_PRINT	Print or copy.
KEY_LL	Home down or bottom (lower left).
KEY_A1	Upper-left key of keypad.
KEY_A3	Upper-right key of keypad.
KEY_B2	Center-key of keypad.
KEY_C1	Lower-left key of keypad.
KEY_C3	Lower-right key of keypad.
KEY_BTAB	Back tab key.
KEY_BEG	beg(inning) key
KEY_CANCEL	cancel key
KEY_CLOSE	close key
KEY_COMMAND	cmd (command) key
KEY_COPY	copy key
KEY_CREATE	create key
KEY_END	end key
KEY_EXIT	exit key
KEY_FIND	find key
KEY_HELP	help key

Item	Description
KEY_MARK	mark key
KEY_MESSAGE	message key
KEY_MOVE	move key
KEY_NEXT	next object key
KEY_OPEN	open key
KEY_OPTIONS	options key
KEY_PREVIOUS	previous object key
KEY_REDO	redo key
KEY_REFERENCE	ref(erence) key
KEY_REFRESH	refresh key
KEY_REPLACE	replace key
KEY_RESTART	restart key
KEY_RESUME	resume key
KEY_SAVE	save key
KEY_SBEG	shifted beginning key

Item	Description
KEY_SCANCEL	shifted cancel key
KEY_SCOMMAND	shifted command key
KEY_SCOPY	shifted copy key
KEY_SCREATE	shifted create key
KEY_SDC	shifted delete char key
KEY_SDL	shifted delete line key
KEY_SELECT	select key
KEY_SEND	shifted end key
KEY_SEOL	shifted clear line key
KEY_SEXIT	shifted exit key
KEY_SFIND	shifted find key
KEY_SHELP	shifted help key
KEY_SHOME	shifted home key
KEY_SIC	shifted input key
KEY_SLEFT	shifted left arrow key
KEY_SMESSAGE	shifted message key
KEY_SMOVE	shifted move key
KEY_SNEXT	shifted next key
KEY_SOPTIONS	shifted options key
KEY_SPREVIOUS	shifted prev key
KEY_SPRINT	shifted print key
KEY_SREDO	shifted redo key
KEY_SREPLACE	shifted replace key
KEY_SRIGHT	shifted right arrow
KEY_SRSUME	shifted resume key
KEY_SSAVE	shifted save key
KEY_SSUSPEND	shifted suspend key
KEY_SUNDO	shifted undo key
KEY_SUSPEND	suspend key
KEY_UNDO	undo key

Parameters

Item	Description
Column	Specifies the horizontal position to move the logical cursor to before getting the character.
Line	Specifies the vertical position to move the logical cursor to before getting the character.
Window	Identifies the window to get the character from and echo it into.

Return Values

Upon successful completion, the **getch, mvwgetch,** and **wgetch** subroutines, CURSES, and Curses Interface return the single-byte character, KEY_ value, or ERR. When in the nodelay mode and no data is available, ERR is returned.

Examples

1. To get a character and echo it to the stdscr, use:

```
mvgetch();
```

2. To get a character and echo it into stdscr at the coordinates y=20, x=30, use:

```
mvgetch(20, 30);
```

3. To get a character and echo it into the user-defined window `my_window` at coordinates y=20, x=30, use:

```
WINDOW *my_window;
mvwgetch(my_window, 20, 30);
```

getcmdattr Subroutine

Purpose

Queries the command security information in the privileged command database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getcmdattr (Command, Attribute, Value, Type)
    char *Command;
    char *Attribute;
    void *Value;
    int Type;
```

Description

The **getcmdattr** subroutine reads a specified attribute from the command database. If the database is not open, this subroutine does an implicit open for reading. For attributes of the **SEC_CHAR** and **SEC_LIST** types, the **getcmdattr** subroutine returns the value to the allocated memory. Caller needs to free this memory.

Parameters

Item	Description
Command	Specifies the command name. The value should be the full path to the command on the system.

Item	Description
Attribute	Specifies the attribute to read. The following possible attributes are defined in the **usersec.h** file: **S_ACCESSAUTHS** Access authorizations. The attribute type is **SEC_LIST** and is a null-separated list of authorization names. Sixteen authorizations can be specified. A user with one of the authorizations is allowed to run the command. In addition to the user-defined and system-defined authorizations available on the system, the following three special values are allowed: **ALLOW_OWNER** Allows the command owner to run the command without checking for access authorizations. **ALLOW_GROUP** Allows the command group to run the command without checking for access authorizations. **ALLOW_ALL** Allows every user to run the command without checking for access authorizations. **S_AUTHPRIVS** Authorized privileges. The attribute type is **SEC_LIST**. Privilege authorization and authorized privileges pairs indicate process privileges during the execution of the command corresponding to the authorization that the parent process possesses. The authorization and its corresponding privileges are separated by an equal sign (=); individual privileges are separated by a plus sign (+); the authorization and privileges pairs are separated by a comma (,) as shown in the following illustration: `auth=priv+priv+...,auth=priv+priv...,...` The number of authorization and privileges pairs is limited to sixteen. **S_AUTHROLES** The role or list of roles, users having these have to be authenticated to allow execution of the command. The attribute type is **SEC_LIST**. **S_INNATEPRIVS** Innate privileges. This is a null-separated list of privileges that are assigned to the process when running the command. The attribute type is **SEC_LIST**. **S_INHERITPRIVS** Inheritable privileges. This is a null-separated list of privileges that are passed to child process privileges. The attribute type is **SEC_LIST**. **S_EUID** The effective user ID to be assumed when running the command. The attribute type is **SEC_INT**. **S_EGID** The effective group ID to be assumed when running the command. The attribute type is **SEC_INT**. **S_RUID** The real user ID to be assumed when running the command. The attribute type is **SEC_INT**.
Value	Specifies a pointer, or a pointer to a pointer according to the value specified in the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Item	Description
Type	Specifies the type of attribute. The following valid types are defined in the **usersec.h** file: **SEC_INT** The format of the attribute is an integer. For the subroutine, the user should supply a pointer to a defined integer variable. **SEC_CHAR** The format of the attribute is a null-terminated character string. For the subroutine, the user should supply a pointer to a defined character pointer variable. Caller needs to free this memory. **SEC_LIST** The format of the attribute is a series of concatenated strings that each of which is null-terminated. The last string in the series is terminated by two successive null characters. For the subroutine, the user should supply a pointer to a defined character pointer variable. Caller needs to free this memory.

Security

Files Accessed:

File	Mode
/etc/security/privcmds	rw

Return Values

If successful, the **getcmdattr** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **getcmdattr** subroutine fails, one of the following **errno** values is set:

Item	Description
EINVAL	The *Command* parameter is **NULL** or **default**.
EINVAL	The *Attribute* array or the *Type* parameter is **NULL** or does not contain one of the defined values.
ENOATTR	The *Attribute* array is **S_PRIVCMDS**, but the *Command* parameter is not **ALL**.
ENOENT	The command specified in the *Command* parameter does not exist.
ENOATTR	The attribute specified in the *Attribute* array is valid, but no value is defined for the command.
EPERM	The operation is not permitted.
EIO	Failed to access remote command database.

getcmdattrs Subroutine

Purpose

Retrieves multiple command attributes from the privileged command database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getcmdattrs(Command, Attributes, Count)
    char *Command;
    dbattr_t *Attributes;
    int Count;
```

Description

The **getcmdattrs** subroutine reads one or more attributes from the privileged command database. The command specified with the *Command* parameter must include the full path to the command and exist in the privileged command database. If the database is not open, this subroutine does an implicit open for reading.

The *Attributes* array contains information about each attribute that is to be read. Each element in the *Attributes* array must be examined upon a successful call to the **getcmdattrs** subroutine to determine whether the *Attributes* array was successfully retrieved. The values of the **SEC_CHAR** or **SEC_LIST** attributes successfully returned are in the allocated memory. Caller need to free this memory after use. The **dbattr_t data** structure contains the following fields:

Item	Description
attr_name	The name of the target command attribute.
attr_idx	This attribute is used internally by the **getcmdattrs** subroutine.
attr_type	The type of the target attribute.
attr _flag	The result of the request to read the target attribute. On successful completion, a value of zero is returned. Otherwise, it returns a nonzero value.
attr_un	A union that contains the returned values for the requested query.
attr_domain	The subroutine ignores any input to this field. If this field is set to null, the subroutine sets this field to the name of the domain where the command is found.

The following valid privileged command attributes for the subroutine are defined in the **usersec.h** file:

Name	Description	Type
S_PRIVCMDS	Retrieves all the commands in the privileged command database. It is valid only when the *Command* parameter is **ALL**.	**SEC_LIST**
S_ACCESSAUTHS	Access authorizations. This is a null-separated list of authorization names. Sixteen authorizations can be specified. A user with any one of the authorizations is allowed to run the command. In addition to the user-defined and system-defined authorizations available on the system, the following three special values are allowed: **ALLOW_OWNER** Allows the command owner to run the command without checking for access authorizations. **ALLOW_GROUP** Allows the command group to run the command without checking for access authorizations. **ALLOW_ALL** Allows every user to run the command without checking for access authorizations.	**SEC_LIST**

Name	Description	Type

S_AUTHPRIVS	Authorized privileges. Privilege authorization and authorized privileges pairs indicate process privileges during the execution of the command corresponding to the authorization that the parent process possesses. The authorization and its corresponding privileges are separated by an equal sign (=); individual privileges are separated by a plus sign (+). The attribute is of the **SEC_LIST** type and the value is a null-separated list, so authorization and privileges pairs are separated by a NULL character (\0), as shown in the following illustration:
	`auth=priv+priv+...`
	`\0auth=priv+priv+...\0...\0\0`
	The number of authorization and privileges pairs is limited to sixteen.
		SEC_LIST

S_AUTHROLES	The role or list of roles, users having these have to be authenticated to allow execution of the command.	**SEC_LIST**
S_INNATEPRIVS	Innate privileges. This is a null-separated list of privileges that are assigned to the process when running the command.	**SEC_LIST**
S_INHERITPRIVS	Inheritable privileges. This is a null-separated list of privileges that are assigned to child processes.	**SEC_LIST**
S_EUID	The effective user ID to be assumed when running the command.	**SEC_INT**
S_EGID	The effective group ID to be assumed when running the command.	**SEC_INT**
S_RUID	The real user ID to be assumed when running the command.	**SEC_INT**

The following union members correspond to the definitions of the **attr_char**, **attr_char**, **attr_int**, **attr_long** and **attr_llong** macros in the **usersec.h** file:

Item	Description
au_char	Attributes of the **SEC_CHAR** and **SEC_LIST** types store a pointer to the returned value in this member when the attributes are successfully retrieved. Caller need to free this memory.
au_int	Storage location for attributes of the **SEC_INT** type.
au_long	Storage location for attributes of the **SEC_LONG** type.
au_llong	Storage location for attributes of the **SEC_LLONG** type.

If **ALL** is specified for the *Command* parameter, the **S_PRIVCMDS** attribute is the only valid attribute that is displayed in the *Attribute* array. Specifying any other attribute with a command name of **ALL** causes the **getcmdattrs** subroutine to fail.

Parameters

Item	Description
Command	Specifies the command for the attributes to be read.
Attributes	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of command attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the *Attributes* array.

Security

Files Accessed:

File	Mode
/etc/security/privcmds	r

Return Values

If the command specified by the *Command* parameter exists in the privileged command database, the **getcmdattrs** subroutine returns zero. On successful completion, the **attr_flag** attribute of each element in the *Attributes* array must be examined to determine whether it was successfully retrieved. On failure, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **getcmdattrs** subroutine returns -1, one of the following **errno** values is set:

Item	Description
EINVAL	The *Command* parameter is **NULL** or **default**.
EINVAL	The *Command* parameter is **ALL** but the Attributes entry contains an attribute other than **S_PRIVCMDS**.
EINVAL	The *Count* parameter is less than zero.
ENOENT	The command specified in the *Command* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.

If the **getcmdattrs** subroutine fails to query an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding attributes element:

Item	Description
EACCES	The invoker does not have access to the attribute that is specified in the **attr_name** field.
EINVAL	The **attr_name** field in the *Attributes* array is not a recognized command attribute.
EINVAL	The **attr_type** field in the *Attributes* array contains a type that is not valid.
EINVAL	The **attr_un** field in the *Attributes* array does not point to a valid buffer.
ENOATTR	The **attr_name** field in the *Attributes* array specifies a valid attribute, but no value is defined for this privileged command.
ENOMEM	Memory cannot be allocated to store the return value.
EIO	Failed to access remote command database.

getconfattr or putconfattr Subroutine

Purpose

Accesses the system information in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <userconf.h>
```

int getconfattr (*sys*, *Attribute*, *Value*, *Type*)
char * *sys*;
char * *Attribute*;
**void ***Value;
int *Type*;

int putconfattr (*sys*, *Attribute*, *Value*, *Type*)
char * *sys*;
char * *Attribute*;
**void ***Value;
int *Type*;

Description

The **getconfattr** subroutine reads a specified attribute from the system information database. The **putconfattr** subroutine writes a specified attribute to the system information database.

Parameters

sys
System attribute. The following possible attributes are defined in the **userconf.h** file.

- SC_SYS_LOGIN
- SC_SYS_USER
- SC_SYS_ADMUSER
- SC_SYS_AUDIT SEC_LIST
- SC_SYS_AUSERS SEC_LIST
- SC_SYS_ASYS SEC_LIST
- SC_SYS_ABIN SEC_LIST
- SC_SYS_ASTREAM SEC_LIST

Users can define the system attribute parameter. In this case, the parameter value is used as a stanza name. The stanza name contains the specified attribute and value in the *Attribute* and *Value* parameters. The `putconfattr` subroutine creates this stanza in the file associated with the attribute. The `getconfattr` subroutine retrieves the value for the specified attribute and user defined stanza.

Attribute
Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

S_CORECOMP
Core compression status. The attribute type is **SEC_CHAR**.

S_COREPATH
Core path specification status. The attribute type is **SEC_CHAR**.

S_COREPNAME
Core path specification location. The attribute type is **SEC_CHAR**.

S_CORENAMING
Core naming status. The attribute type is **SEC_CHAR**.

S_PGRP
Principle group name.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the Lightweight Directory Access Protocol (LDAP) group can be assigned to LOCAL user as primary group and vice versa.

The attribute type is **SEC_CHAR**.

S_GROUPS
Groups to which the user belongs.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_ADMGROUPS
Groups for which the user is an administrator.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_ADMIN
Administrative status of a user. The attribute type is **SEC_BOOL**.

S_AUDITCLASSES
Audit classes to which the user belongs. The attribute type is **SEC_LIST**.

S_AUTHSYSTEM
Defines the user's authentication method. The attribute type is **SEC_CHAR.**

S_HOME
Home directory. The attribute type is **SEC_CHAR**.

S_SHELL
Initial program run by a user. The attribute type is **SEC_CHAR**.

S_GECOS
Personal information for a user. The attribute type is **SEC_CHAR**.

S_USRENV
User-state environment variables. The attribute type is **SEC_LIST**.

S_SYSENV
Protected-state environment variables. The attribute type is **SEC_LIST**.

S_LOGINCHK
Specifies whether the user account can be used for local logins. The attribute type is **SEC_BOOL**.

S_HISTEXPIRE
Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is **SEC_INT**.

S_HISTSIZE
Specifies the number of previous passwords that the user cannot reuse. The attribute type is **SEC_INT**.

S_MAXREPEAT
Defines the maximum number of times a user can repeat a character in a new password. The attribute type is **SEC_INT**.

S_MINAGE
Defines the minimum age in weeks that the user's password must exist before the user can change it. The attribute type is **SEC_INT**.

S_PWDCHECKS
Defines the password restriction methods for this account. The attribute type is **SEC_LIST**.

S_MINALPHA
Defines the minimum number of alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_MINDIFF
Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is **SEC_INT**.

S_MINLEN
Defines the minimum length of a user's password. The attribute type is **SEC_INT**.

S_MINOTHER
Defines the minimum number of non-alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_DICTIONLIST
Defines the password dictionaries for this account. The attribute type is **SEC_LIST**.

S_SUCHK
Specifies whether the user account can be accessed with the **su** command. Type **SEC_BOOL**.

S_REGISTRY
Defines the user's authentication registry. The attribute type is **SEC_CHAR**.

S_RLOGINCHK
Specifies whether the user account can be used for remote logins using the **telnet** or **rlogin** commands. The attribute type is **SEC_BOOL**.

S_DAEMONCHK
Specifies whether the user account can be used for daemon execution of programs and subsystems using the **cron** daemon or **src**. The attribute type is **SEC_BOOL**.

S_TPATH
Defines how the account may be used on the trusted path. The attribute type is **SEC_CHAR**. This attribute must be one of the following values:

nosak
The secure attention key is not enabled for this account.

notsh
The trusted shell cannot be accessed from this account.

always
This account may only run trusted programs.

on
Normal trusted-path processing applies.

S_MINLOWERALPHA
Defines the minimum number of lowercase alphabetic characters required in a new user password. The attribute type is **SEC_INT**.

S_MINUPPERALPHA
Defines the minimum number of uppercase alphabetic characters required in a new user password. The attribute type is **SEC_INT**.

S_MINDIGIT
Defines the minimum number of digits required in a new user password. The attribute type is **SEC_INT**.

S_MINSPECIALCHAR
Defines the minimum number of special characters required in a new user password. The attribute type is **SEC_INT**.

S_TTYS
List of ttys that can or cannot be used to access this account. The attribute type is **SEC_LIST**.

S_SUGROUPS
Groups that can or cannot access this account.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_EXPIRATION
Expiration date for this account, in seconds since the epoch. The attribute type is **SEC_CHAR**.

S_AUTH1
Primary authentication methods for this account. The attribute type is **SEC_LIST**.

S_AUTH2
Secondary authentication methods for this account. The attribute type is **SEC_LIST**.

S_UFSIZE
Process file size soft limit. The attribute type is **SEC_INT**.

S_UCPU
Process CPU time soft limit. The attribute type is **SEC_INT**.

S_UDATA
Process data segment size soft limit. The attribute type is **SEC_INT**.

S_USTACK
Process stack segment size soft limit. Type: **SEC_INT**.

S_URSS
Process real memory size soft limit. Type: **SEC_INT**.

S_UCORE
Process core file size soft limit. The attribute type is **SEC_INT**.

S_PWD
Specifies the value of the passwd field in the **/etc/passwd** file. The attribute type is **SEC_CHAR**.

S_UMASK
File creation mask for a user. The attribute type is **SEC_INT**.

S_LOCKED
Specifies whether the user's account can be logged into. The attribute type is **SEC_BOOL**.

S_UFSIZE_HARD
Process file size hard limit. The attribute type is **SEC_INT**.

S_UCPU_HARD
Process CPU time hard limit. The attribute type is **SEC_INT**.

S_UDATA_HARD
Process data segment size hard limit. The attribute type is **SEC_INT**.

S_USTACK_HARD
Process stack segment size hard limit. Type: **SEC_INT**.

S_URSS_HARD
Process real memory size hard limit. Type: **SEC_INT**.

S_UCORE_HARD
Process core file size hard limit. The attribute type is **SEC_INT**.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations.

Type
Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include:

SEC_INT
The format of the attribute is an integer.

For the **getconfattr** subroutine, the user should supply a pointer to a defined integer variable. For the **putconfattr** subroutine, the user should supply an integer.

SEC_CHAR
The format of the attribute is a null-terminated character string.

SEC_LIST
The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters.

SEC_BOOL
The format of the attribute from the **getconfattr** subroutine is an integer with the value of either 0 (false) or 1 (true). The format of the attribute for the **putconfattr** subroutine is a null-terminated string containing one of the following strings: true, false, yes, no, always, or never.

SEC_COMMIT
For the **putconfattr** subroutine, this value specified by itself indicates that the changes to the named *sys* value or stanza are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no stanza name is specified, all outstanding changes to the system information databases are committed to permanent storage.

SEC_DELETE
The corresponding attribute is deleted from the database.

Security

Item	Description

Files Accessed:

Mode	File
rw	**/etc/security/user**
rw	**/etc/security/limits**
rw	**/etc/security/login.cfg**
rw	**/usr/lib/security/mkuser.default**
rw	**/etc/security/audit/config**

Return Values

If successful, the **getconfattr** subroutine returns a value of zero.

If unsuccessful, the **getconfattr** subroutine returns a value of -1.

Error Codes

Item	Description
ENOENT	The value that the *Sys* parameter specifies does not exist.
ENOATTR	The specified *Attribute* variable is not defined for this *Sys* parameter.
EINVAL	The *Attribute* or *Type* variable for the specified *Sys* parameter is not valid.
EACCESS	The user does not have access to the specified *Attribute* variable.
EIO	Failed to access remote system information database.

Files

Item	Description
/etc/passwd	Contains user IDs.

getconfattrs Subroutine

Purpose

Accesses system information in the system information database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <userconf.h>
```

int getconfattrs (*Sys*, *Attributes*, *Count*)
char * *Sys*;
dbattr_t * *Attributes*;
int *Count*

Description

The **getconfattrs** subroutine accesses system configuration information.

The **getconfattrs** subroutine reads one or more attributes from the system configuration database. If the database is not already open, this subroutine does an implicit open for reading.

The *Attributes* array contains information about each attribute that is to be written. The **dbattr_t** data structure contains the following fields:

attr_name
The name of the desired attribute.

attr_idx
Used internally by the **getconfattrs** subroutine.

attr_type
The type of the desired attribute. The list of attribute types is defined in the **usersec.h** header file.

attr_flag
The results of the request to read the desired attribute.

attr_un
A union containing the values to be written. Its union members that follow correspond to the definitions of the **attr_char**, **attr_int**, **attr_long**, and **attr_llong** macros, respectively:

au_char
Attributes of type **SEC_CHAR** and **SEC_LIST** store a pointer to the value to be written.

au_int
Attributes of type **SEC_INT** and **SEC_BOOL** contain the value of the attribute to be written.

au_long
Attributes of type **SEC_LONG** contain the value of the attribute to be written.

au_llong
Attributes of type **SEC_LLONG** contain the value of the attribute to be written.

attr_domain
The authentication domain containing the attribute. The **getconfattrs** subroutine is responsible for managing the memory referenced by this pointer.

Use the **setuserdb** and **enduserdb** subroutines to open and close the system configuration database. Failure to explicitly open and close the system database can result in loss of memory and performance.

Parameters

Item	Description
Sys	Specifies the name of the subsystem for which the attributes are to be read.
Attributes	A pointer to an array of one or more elements of type **dbattr_t**. The list of system attributes is defined in the **usersec.h** header file.
Count	The number of array elements in *Attributes*.

Security

Files accessed:

Item	Description
Mode	**File**
r	**/etc/security/.ids**
r	**/etc/security/audit/config**
r	**/etc/security/audit/events**
r	**/etc/security/audit/objects**
r	**/etc/security/login.cfg**
r	**/etc/security/portlog**
r	**/etc/security/roles**
r	**/usr/lib/security/methods.cfg**
r	**/usr/lib/security/mkuser.default**

Return Values

If the value of the *Sys* or *Attributes* parameter is NULL, or the value of the *Count* parameter is less than 1, the **getconfattrs** subroutine returns a value of -1, and sets the **errno** global variable to indicate the error. Otherwise, the subroutine returns a value of zero. The **getconfattrs** subroutine does not check the validity of the *Sys* parameter. Each element in the *Attributes* array must be examined on a successful call to the **getconfattrs** subroutine to determine whether the *Attributes* array entry is successfully retrieved.

Error Codes

The **getconfattrs** subroutine returns an error that indicates that the system attribute does or does not exist. Additional errors can indicate an error querying the information databases for the requested attributes.

Item	Description
EINVAL	The *Attributes* parameter is NULL.
EINVAL	The *Count* parameter is less than 1.
ENOENT	The specified *Sys* does not exist.
EIO	Failed to access remote system information database.

If the **getconfattrs** subroutine fails to query an attribute, one or more of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The user does not have access to the attribute specified in the **attr_name** field.

Item	Description
EINVAL	The **attr_type** field in the *Attributes* entry contains an invalid type.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
ENOMEM	Memory could not be allocated to store the return value.
ENOATTR	The **attr_name** field in the *Attributes* entry specifies an attribute that is not defined for this system table.

Files

Item	Description
/etc/security/.ids	The next available user and group ID values.
/etc/security/audit/config	Bin and stream mode audit configuration parameters.
/etc/security/audit/events	Format strings for audit event records.
/etc/security/audit/objects	File system objects that are explicitly audited.
/etc/security/login.cfg	Miscellaneous login relation parameters.
/etc/security/portlog	Port login failure and locking history.
/etc/security/roles	Definitions of administrative roles.
/usr/lib/security/methods.cfg	Definitions of loadable authentication modules.
/usr/lib/security/mkuser.default	Default user attributes for administrative and non administrative users.

getcontext or setcontext Subroutine

Purpose

Initializes the structure pointed to by *ucp* to the context of the calling process.

Library

(**libc.a**)

Syntax

#include <ucontext.h>

int getcontext (ucontext_t ****ucp*);**

**int setcontext (const uncontext_t ****ucp*);

Description

The **getcontext** subroutine initalizes the structure pointed to by *ucp* to the current user context of the calling process. The **ucontext_t** type that *ucp* points to defines the user context and includes the contents of the calling process' machine registers, the signal mask, and the current execution stack.

The setcontext subroutine restores the user context pointed to by *ucp*. A successful call to setcontext subroutine does not return; program execution resumes at the point specified by the *ucp* argument passed to setcontext subroutine. The *ucp* argument should be created either by a prior call to getcontext subroutine, or by being passed as an argument to a signal handler. If the *ucp* argument was created with

getcontext subroutine, program execution continues as if the corresponding call of getcontext subroutine had just returned. If the *ucp* argument was created with makecontext subroutine, program execution continues with the function passed to makecontext subroutine. When that function returns, the process continues as if after a call to setcontext subroutine with the *ucp* argument that was input to makecontext subroutine. If the *ucp* argument was passed to a signal handler, program execution continues with the program instruction following the instruction interrupted by the signal. If the uc_link member of the ucontext_t structure pointed to by the *ucp* arguement is equal to 0, then this context is the main context, and the process will exit when this context returns.

Parameters

Item	Description
ucp	A pointer to a user stucture.

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getcontext** and **setcontext** subroutines are unsuccessful if one of the following is true:

Item	Description
EINVAL	NULL *ucp* address
EFAULT	Invalid *ucp* address

getcwd Subroutine

Purpose

Gets the path name of the current directory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

char *getcwd ( Buffer,  Size)
char *Buffer;
size_t Size;
```

Description

The **getcwd** subroutine places the absolute path name of the current working directory in the array pointed to by the *Buffer* parameter, and returns that path name. The *size* parameter specifies the size in bytes of the character array pointed to by the *Buffer* parameter.

Parameters

Item	Description
Buffer	Points to string space that will contain the path name. If the *Buffer* parameter value is a null pointer, the **getcwd** subroutine, using the **malloc** subroutine, obtains the number of bytes of free space as specified by the *Size* parameter. In this case, the pointer returned by the **getcwd** subroutine can be used as the parameter in a subsequent call to the **free** subroutine. Starting the **getcwd** subroutine with a null pointer as the *Buffer* parameter value is not recommended.

Item	Description
Size	Specifies the length of the string space. The value of the *Size* parameter must be at least 1 greater than the length of the path name to be returned.

Return Values

If the **getcwd** subroutine is unsuccessful, a null value is returned and the **errno** global variable is set to indicate the error. The **getcwd** subroutine is unsuccessful if the *Size* parameter is not large enough or if an error occurs in a lower-level function.

Error Codes

If the **getcwd** subroutine is unsuccessful, it returns one or more of the following error codes:

Item	Description
EACCES	Indicates that read or search permission was denied for a component of the path name
EINVAL	Indicates that the *Size* parameter is 0 or a negative number.
ENOMEM	Indicates that insufficient storage space is available.
ERANGE	Indicates that the *Size* parameter is greater than 0, but is smaller than the length of the path name plus 1.

getdate Subroutine

Purpose

Convert user format date and time.

Library

Standard C Library **(libc.a)**

Syntax

```
#include <time.h>

struct tm *getdate (const char *string)
extern int getdate_err
```

Description

The **getdate** subroutine converts user definable date and/or time specifications pointed to by *string*, into a **struct tm**. The structure declaration is in the **time.h** header file (see **ctime** subroutine).

User supplied templates are used to parse and interpret the input string. The templates are contained in text files created by the user and identified by the environment variable **DATEMSK**. The **DATEMSK** variable should be set to indicate the full pathname of the file that contains the templates. The first line in the template that matches the input specification is used for interpretation and conversation into the internal time format.

The templates should follow a format where complex field descriptors are preceded by simpler ones. For example:

```
%M
%H:%M
%m/%d/%y
%m/%d/%y %H:%M:%S
```

The following field descriptors are supported:

Item	Description
%%	Same as %.
%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%c	Locale's appropriate date and time representation.
%C	Century number (00-99; leading zeros are permitted but not required)
%d	Day of month (01 - 31: the leading zero is optional.
%e	Same as %d.
%D	Date as %m/%d/%y.
%h	Abbreviated month name.
%H	Hour (00 - 23)
%I	Hour (01 - 12)
%m	Month number (01 - 12)
%M	Minute (00 - 59)
%n	Same as \n.
%p	Locale's equivalent of either AM or PM.
%r	Time as %I:%M:%S %p
%R	Time as %H: %M
%S	Seconds (00 - 61) Leap seconds are allowed but are not predictable through use of algorithms.
%t	Same as tab.
%T	Time as %H: %M:%S
%w	Weekday number (Sunday = 0 - 6)
%x	Locale's appropriate date representation.
%X	Locale's appropriate time representation.

Item	Description
%y	Year within century. **Note:** When the environment variable **XPG_TIME_FMT=ON**, **%y** is the year within the century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999, inclusive); values in the range 00-68 refer to 2000 to 2068, inclusive.
%Y	Year as ccyy (such as 1986)
%Z	Time zone name or no characters if no time zone exists. If the time zone supplied by %Z is not the same as the time zone **getdate** subroutine expects, an invalid input specification error will result. The **getdate** subroutine calculates an expected time zone based on information supplied to the interface (such as hour, day, and month).

The match between the template and input specification performed by the **getdate** subroutine is case sensitive.

The month and weekday names can consist of any combination of upper and lower case letters. The used can request that the input date or time specification be in a specific language by setting the LC_TIME category (See the **setlocale** subroutine).

Leading zero's are not necessary for the descriptors that allow leading zero's. However, at most two digits are allowed for those descriptors, including leading zero's. Extra whitespace in either the template file or in *string* is ignored.

The field descriptors %c, %x, and %X will not be supported if they include unsupported field descriptors.

Example 1 is an example of a template. Example 2 contains valid input specifications for the template. Example 3 shows how local date and time specifications can be defined in the template.

The following rules apply for converting the input specification into the internal format:

- If only the weekday is given, today is assumed if the given month is equal to the current day and next week if it is less.
- If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given (the first day of month is assumed if no day is given).
- If no hour, minute, and second are given, the current hour, minute and second are assumed.
- If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

Return Values

Upon successful completion, the **getdate** subroutine returns a pointer to **struct tm;** otherwise, it returns a null pointer and the external variable **getdate_err** is set to indicate the error.

Error Codes

Upon failure, a null pointer is returned and the variable **getdate_err** is set to indicate the error.

The following is a complete list of the **getdate_err** settings and their corresponding descriptions:

Item	Description
1	The **DATEMSK** environment variable is null or undefined.
2	The template file cannot be opened for reading.
3	Failed to get file status information.
4	The template file is not a regular file.

Item	Description
5	An error is encountered while reading the template file.
6	Memory allocation failed (not enough memory available.
7	There is no line in the template that matches the input.
8	Invalid input specification, Example: February 31 or a time is specified that can not be represented in a time_t (representing the time in seconds since 00:00:00 UTC, January 1, 1970).

Examples

1. The following example shows the possible contents of a template:

```
%m
%A %B %d, %Y, %H:%M:%S
%A
%B
%m/%d/%y %I %p
%d, %m, %Y %H:%M
at %A the %dst of %B in %Y
run job at %I %p, %B %dnd
&A den %d. %B %Y %H.%M Uhr
```

2. The following are examples of valid input specifications for the template in Example 1:

```
getdate ("10/1/87 4 PM")
getdate ("Friday")
getdate ("Friday September 18, 1987, 10:30:30")
getdate ("24,9,1986 10:30")
getdate ("at monday the 1st of december in 1986")
getdate ("run job at 3 PM. december 2nd")
```

   If the LC_TIME category is set to a German locale that includes `freitag` as a weekday name and `oktober` as a month name, the following would be valid:

```
getdate ("freitag den 10. oktober 1986 10.30 Uhr")
```

3. The following examples shows how local date and time specification can be defined in the template.

Invocation	Line in Template
getdate ("11/27/86")	%m/%d/%y
getdate ("27.11.86"0	%d.%m.%y
getdate ("86-11-27")	%y-%m-%d
getdate ("Friday 12:00:00")	%A %H:%M:%S

4. The following examples help to illustrate the above rules assuming that the current date Mon Sep 22 12:19:47 EDT 1986 and the LC_TIME category is set to the default "C" locale.

Input	Line in Template	Date
Mon	%a	Mon Sep 22 12:19:47 EDT 1986
Sun	%a	Sun Sep 28 12:19:47 EDT 1986
Fri	%a	Fri Sep 26 12:19:47 EDT 1986
September	%B	Mon Sep1 12:19:47 EDT 1986
January	%B	Thu Jan 1 12:19:47 EDT 1986
December	%B	Mon Dec 1 12:19:47 EDT 1986
Sep Mon	%b %a	Mon Sep 1 12:19:47 EDT 1986

Input	Line in Template	Date
Jan Fri	%b %a	Fri Jan 2 12:19:47 EDT 1986
Dec Mon	%b %a	Mon Dec 1 12:19:47 EDT 1986
Jan Wed 1989	%b %a %Y	Wed Jan 4 12:19:47 EDT 1986
Fri 9	%a %H	Fri Sep 26 12:19:47 EDT 1986
Feb 10:30	%b %H: %S	Sun Feb 1 12:19:47 EDT 1986
10:30	%H: %M	Tue Sep 23 12:19:47 EDT 1986
13:30	%H: %M	Mon Sep 22 12:19:47 EDT 1986

getdevattr Subroutine

Purpose

Retrieves the device security information in the privileged device database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getdevattr (Device, Attribute, Value, Type)
    char *Device;
    char *Attribute;
    void *Value;
    int Type;
```

Description

The **getdevattr** subroutine reads a specified attribute from the device database. If the database is not open, this subroutine does an implicit open for reading. For attributes of the **SEC_CHAR** and **SEC_LIST** types, the **getdevattr** subroutine returns the value to the allocated memory. Caller needs to free this memory.

Parameters

Item	Description
Device	Specifies the device name. The value should be the full path to the device on the system. This parameter must be specified unless the *Type* parameter is **SEC_COMMIT**.
Attribute	Specifies the attribute that is read. The following possible attributes are defined in the **usersec.h** file: **S_READPRIVS** Privileges required to read from the device. Eight privileges can be defined. A process with any of the read privileges is allowed to read from the device. The attribute type is **SEC_LIST**. **S_WRITEPRIVS** Privileges required to write to the device. Eight privileges can be defined. A process with any of the write privileges is allowed to write to the device.

Item	Description
Value	Specifies a pointer or a pointer to a pointer according to the *Attribute* array and the *Type* parameters. See the *Type* parameter for more details.
Type	Specifies the type of attribute. The following valid types are defined in the **usersec.h** file: **SEC_INT** The format of the attribute is an integer. For the **getdevattr** subroutine, the user should supply a pointer to a defined integer variable. **SEC_CHAR** The format of the attribute is a null-terminated character string. For the **getdevattr** subroutine, the user should supply a pointer to a defined character pointer variable. The value is returned as allocated memory for the **getdevattr** subroutine. Caller need to free this memory. **SEC_LIST** The format of the attribute is a series of concatenated strings, each of which is null-terminated. The last string in the series is terminated by two successive null characters. For the **getdevattr** subroutine, the user should supply a pointer to a defined character pointer variable. Caller need to free this memory.

Security

Files Accessed:

File	Mode
/etc/security/privdevs	rw

Return Values

On successful completion, the **getdevattr** subroutine returns a value of zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **getdevattr** subroutine fails, one of the following **errno** values is set:

Item	Description
EINVAL	The *Device* parameter is **NULL** or **default**.
EINVAL	The *Attribute* or *Type* parameter is **NULL** or does not contain one of the defined values.
EINVAL	The *Attribute* parameter is **S_PRIVDEVS**, but the *Device* parameter is not **ALL**.
ENOENT	The device specified in the *Device* parameter does not exist.
ENOATTR	The attribute specified in the *Attribute* parameter is valid, but no value is defined for the device.
EPERM	The operation is not permitted.

getdevattrs Subroutine

Purpose

Retrieves multiple device attributes from the privileged device database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getdevattrs(Device, Attributes, Count)
    char *Device;
    dbattr_t *Attributes;
    int Count;
```

Description

The **getdevattrs** subroutine reads one or more attributes from the privileged device database. The device specified with the *Device* parameter must include the full path to the device and exist in the privileged device database. If the database is not open, this subroutine does an implicit open for reading.

The *Attributes* parameter contains information about each attribute that is to be read. Each element in the *Attributes* parameter must be examined on a successful call to the **getdevattrs** subroutine to determine whether the *Attributes* parameter was successfully retrieved. The values of the **SEC_CHAR** or **SEC_LIST** attributes that are successfully returned are in the allocated memory. Caller need to free this memory after use. The **dbattr_t data** structure contains the following fields:

Item	Description
attr_name	The name of the target device attribute.
attr_idx	This attribute is used internally by the **getdevattrs** subroutine.
attr_type	The type of the target attribute.
attr _flag	The result of the request to read the target attribute. On successful completion, the value of zero is returned. Otherwise, a nonzero value is returned.
attr_un	A union that contains the returned values for the requested query.
attr_domain	The subroutine ignores any input to this field. If this field is set to null, the subroutine sets this field to the name of the domain where the device is found.

The following valid privileged device attributes for the **getdevattrs** subroutine are defined in the **usersec.h** file:

Name	Description	Type
S_PRIVDEVS	Retrieves all the devices in the privileged device database. It is valid only when the *Device* parameter is set to **ALL**.	**SEC_LIST**
S_READPRIVS	The privileges that are required to read from the device. Eight privileges can be defined. A process with any of the read privileges is allowed to read from the device.	**SEC_LIST**

Name	Description	Type
S_WRITEPRIVS	The privileges that are required to write to the device. Eight privileges can be defined. A process with any of the write privileges is allowed to write to the device.	**SEC_LIST**

The following union members correspond to the definitions of the **attr_char**, **attr_init**, **attr_long** and the **attr_llong** macros in the **usersec.h** file respectively.

Item	Description
au_char	The attributes of the **SEC_CHAR** and **SEC_LIST** types store a pointer to the returned value in this member when the attributes are successfully retrieved. Caller need to free this memory.
au_int	The storage location for attributes of the **SEC_INT** type.
au_long	The storage location for attributes of the **SEC_LONG** type.
au_llong	The storage location for attributes of the **SEC_LLONG** type.

If **ALL** is specified for the *Device* parameter, the **S_PRIVDEVS** attribute is the only valid attribute that is displayed in the *Attribute* parameter. Specifying any other attribute with a device name of **ALL** causes the **getdevattrs** subroutine to fail.

Parameters

Item	Description
Device	Specifies the device for which the attributes are to be read.
Attributes	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of device attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the *Attributes* parameter.

Security

Files Accessed:

File	Mode
/etc/security/privdevs	r

Return Values

If the device that is specified by the *Device* parameter exists in the privileged device database, the **getdevattrs** subroutine returns zero. On successful completion, the **attr_flag** attribute of each element in the *Attributes* parameter must be examined to determine whether it was successfully retrieved. On failure, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **getdevattrs** subroutine returns -1, one of the following **errno** values is set:

Item	Description
EINVAL	The *Device* parameter is **NULL** or **default**.
EINVAL	The *Device* parameter is **ALL**, but the *Attributes* parameter contains an attribute other than **S_PRIVDEVS**.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Device* parameter is **NULL** and the *Count* parameter is greater than zero.
ENOENT	The device specified in the *Device* parameter does not exist.
EPERM	The operation is not permitted.

If the **getdevattrs** subroutine fails to query an attribute, one of the following errors is returned to the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The invoker does not have access to the attribute specified in the **attr_name** field.
EINVAL	The **attr_name** field in the *Attributes* parameter is not a recognized device attribute.
EINVAL	The **attr_type** field in the *Attributes* parameter contains a type that is not valid.
EINVAL	The **attr_un** field in the *Attributes* parameter does not point to a valid buffer.
ENOATTR	The **attr_name** field in the *Attributes* parameter specifies a valid attribute, but no value is defined for this device.
ENOMEM	Memory cannot be allocated to store the return value.

getdomattr Subroutine

Purpose

Queries the domains that are defined in the domain database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int getdomattr ( Dom, Attribute, Value, Type)
char * Domain;
char * Attribute;

void *Value;
int Type;
```

Description

The **getdomattr** subroutine reads a specified attribute from the domain database. If the database is not open, this subroutine does an implicit open for reading. For the attributes of the SEC_CHAR and SEC_LIST types, the **getdomattr** subroutine returns the value to the allocated memory. The caller must free this memory.

Parameters

Item	Description
Dom	Specifies the domain name.
Attribute	Specifies the attribute to read. The following possible attributes are defined in the **usersec.h** file: S_DOMAINS A list of all available domains on the system. This attribute is read only and is only available to the **getdomattr** subroutine when ALL is specified for the Dom parameter. The attribute type is SEC_LIST. S_ID Specifies a unique integer that is used to identify the domains. The attribute type is SEC_INT. S_DFLTMSG Specifies the default domain description to use if message catalogs are not in use. The attribute type is SEC_CHAR. S_MSGCAT Specifies the message catalog file name that contains the description of the domain . The attribute type is SEC_CHAR. S_MSGSET Specifies the message set that contains the description of the domain in the file that the S_MSGCAT attribute specifies. The attribute type is SEC_INT. S_MSGNUMBER Specifies the message number for the description of the domain in the file that the S_MSGCAT attribute specifies and the message set that the S_MSGSET attribute specifies. The attribute type is SEC_INT.
Value	Specifies a pointer, or a pointer to a pointer according to the value specified in the *Attribute* and *Type* parameters. See the *Type* parameter for more details. Specifies the type of attribute. The following valid types are defined in the **usersec.h** file: SEC_INT The format of the attribute is an integer. For the subroutine, the user should supply a pointer to a defined integer variable.
Type	SEC_LIST The format of the attribute is a series of concatenated strings that each of which is null-terminated. The last string in the series is terminated by two successive null characters. For the subroutine, the user should supply a pointer to a defined character pointer variable. Caller needs to free this memory.

Security

Files Accessed:

Item	Description
File	**Mode**

Item	Description
/etc/security/domains	R

Return Values

If successful, the **getdomattr** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

Item	Description
EINVAL	The *Dom* parameter is NULL. The *Attribute* or *Type* parameter is NULL or does not contain one of the defined values. The *Dom* parameter is ALL and the *Attribute* parameter is not S_DOMAINS. The *Value* parameter does not point to a valid buffer for this type of attribute.
ENOATTR	The *Attribute* parameter is S_DOMAINS, but the *Dom* parameter is not ALL The attribute specified in the Attribute parameter is valid but no value is defined for the domain. .
ENOENT	The domain specified in the *Dom* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

getdomattrs Subroutine

Purpose

Retrieves multiple domain attributes from the domain-assigned object database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int getdomattrs ( Dom, Attributes, Count)
char * Dom;
dbattr_t * Attributes;
int Count;
```

Description

The **getdomattrs** subroutine reads one or more attributes from the domain-assigned object database. The *Attributes* array contains information about each attribute that is to be read. Each element in the Attributes array must be examined upon a successful call to the **getdomattrs** subroutine, to determine

whether the *Attributes* array was successfully retrieved. The attributes of the SEC_CHAR or SEC_LIST type will have their values returned to the allocated memory. The caller need to free this memory. The **dbattr_t** data structure contains the following fields:

Item	Description
attr_name	The name of the target domain attribute.
attr_idx	This attribute is used internally by the **getdomattrs** subroutine.
attr_type	The type of a target attribute.
attr _flag	The result of the request to read the target attribute. On successful completion, a value of zero is returned. Otherwise, a value of nonzero is returned.
attr_un	A union that contains the returned values for the requested query.
attr_domain	The **getdomattrs** subroutine ignores any input to this field. If this field is set to null, the subroutine sets this field to the name of the domain where the domain is found.

The following valid domain attributes for the **getdomattrs** subroutine are defined in the **usersec.h** file:

Name	Description	Type
S_DOMAINS	A list of all available domains on the system. It is valid only when the *Dom* parameter is set to the ALL variable.	SEC_LIST
S_DFLTMSG	The default domain description that is used when catalogs are not in use.	SEC_CHAR
S_ID	A unique integer that is used to identify the domain.	SEC_INT
S_MSGCAT	The message catalog name that contains the domain description.	SEC_CHAR
S_MSGSET	The message catalog set number of the domain description.	SEC_INT
S_MSGNUMBER	The message number of the domain description.	SEC_INT

The following union members correspond to the definitions of the ATTR_CHAR, ATTR_INT, ATTR_LONG and ATTR_LLONG macros in the **usersec.h** file:

Item	Description
au_char	Attributes of the SEC_CHAR and SEC_LIST types store a pointer to the returned value in this member when the attributes are successfully retrieved. The caller is responsible for freeing this memory.
au_int	The storage location for attributes of the SEC_INT type.

Item	Description
au_long	The storage location for attributes of the SEC_LONG type.
au_llong	The storage location for attributes of the SEC_LLONG type.

If ALL is specified for the *Dom* parameter, the only valid attribute that can be displayed in the Attribute array is the S_DOMAINS attribute. Specifying any other attribute with an domain name of ALL causes the **getdomattrs** subroutine to fail.

Parameters

Item	Description
Dom	Specifies the domain name for the *Attributes* array to read.
Attribute	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of domain attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the *Attributes* array.

Security

Files Accessed:

Item	Description
File	**Mode**
`/etc/security/domains`	`r`

Return Values

If the domain that is specified by the *Dom* parameter exists in the domain database, the **getdomattrs** subroutine returns the value of zero. On successful completion, the **attr_flag** attribute of each element in the *Attributes* array must be examined to determine whether it was successfully retrieved. If the specified domain does not exist, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

Item	Description
EINVAL	The *Dom* parameter is NULL. The *Count* parameter is less than zero. The *Attributes* array is NULL and the Count parameter is greater than zero. The *Dom* parameter is ALL but the Attributes entry contains an attribute other than S_DOMAINS.
ENOENT	The domain specified in the *Dom* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

If the **getdomattrs** subroutine fails to query an attribute, one of the following errors is returned to the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The invoker does not have access to the attribute specified in the **attr_name** field.
EINVAL	The **attr_name** field in the *Attributes* entry is not a recognized domain attribute. The **attr_type** field in the *Attributes* entry contains a type that is not valid. The **attr_un** field in the *Attributes* entry does not point to a valid buffer.
ENOATTR	The **attr_name** field in the *Attributes* entry specifies a valid attribute, but no value is defined for this domain.

getdtablesize Subroutine

Purpose

Gets the descriptor table size.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int getdtablesize (void)
```

Description

The **getdtablesize** subroutine is used to determine the size of the file descriptor table.

The size of the file descriptor table for a process is set by the **ulimit** command or by the **setrlimit** subroutine. The **getdtablesize** subroutine returns the current size of the table as reported by the **getrlimit** subroutine. If **getrlimit** reports that the table size is unlimited, **getdtablesize** instead returns the value of OPEN_MAX, which is the largest possible size of the table.

Note: The **getdtablesize** subroutine returns a runtime value that is specific to the version of the operating system on which the application is running. The **getdtablesize** returns a value that is set in the **limits** file, which can be different from system to system.

Return Values

The **getdtablesize** subroutine returns the size of the descriptor table.

getea Subroutine

Purpose

Reads the value of an extended attribute.

Syntax

```
#include <sys/ea.h>

ssize_t getea(const char *path, const char *name,
        void *value, size_t size);
ssize_t fgetea(int filedes, const char *name, void *value, size_t size);
ssize_t lgetea(const char *path, const char *name,
        void *value, size_t size);
```

Description

Extended attributes are name:value pairs associated with the file system objects (such as files, directories, and symlinks). They are extensions to the normal attributes that are associated with all objects in the file system (that is, the **stat(2)** data).

Do not define an extended attribute name with the eight characters prefix "(0xF8)SYSTEM(0xF8)". Prefix "(0xF8)SYSTEM(0xF8)" is reserved for system use only.

Note: The 0xF8 prefix represents a non-printable character.

The **getea** subroutine retrieves the value of the extended attribute identified by *name* and associated with the given *path* in the file system. The length of the attribute *value* is returned. The **fgetea** subroutine is identical to **getea**, except that it takes a file descriptor instead of a path. The **lgetea** subroutine is identical to **getea**, except, in the case of a symbolic link, the link itself is interrogated rather than the file that it refers to.

Parameters

Item	Description
path	The path name of the file.
name	The name of the extended attribute. An extended attribute name is a NULL-terminated string.
value	A pointer to a buffer in which the attribute will be stored. The value of an extended attribute is an opaque byte stream of specified length.
size	The size of the buffer. If size is 0, **getea** returns the current size of the named extended attribute, which can be used to estimate whether the size of a buffer is sufficiently large enough to hold the value associated with the extended attribute.
filedes	A file descriptor for the file.

Return Values

If the **getea** subroutine succeeds, a nonnegative number is returned that indicates the size of the extended attribute value. Upon failure, -1 is returned and **errno** is set appropriately.

Error Codes

Item	Description
EACCES	Caller lacks read permission on the base file, or lacks the appropriate ACL privileges for named attribute **read**.
EFAULT	A bad address was passed for *path*, *name*, or *value*.
EFORMAT	File system is capable of supporting EAs, but EAs are disabled.
EINVAL	A path-like name should not be used (such as **zml/file**, **.** and **..**).
ENAMETOOLONG	The *path* or *name* value is too long.

Item	Description
ENOATTR	The named attribute does not exist, or the process has no access to this attribute.
ERANGE	The size of the value buffer is too small to hold the result.
ENOTSUP	Extended attributes are not supported by the file system.

The errors documented for the **stat(2)** system call are also applicable here.

getenv Subroutine

Purpose

Returns the value of an environment variable.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

char *getenv (*Name*)
const char *Name;

Description

The **getenv** subroutine searches the environment list for a string of the form *Name*=*Value*. Environment variables are sometimes called shell variables because they are frequently set with shell commands.

Parameters

Item	Description
Name	Specifies the name of an environment variable. If a string of the proper form is not present in the current environment, the **getenv** subroutine returns a null pointer.

Return Values

The **getenv** subroutine returns a pointer to the value in the current environment, if such a string is present. If such a string is not present, a null pointer is returned. The **getenv** subroutine normally does not modify the returned string. The **putenv** subroutine, however, may overwrite or change the returned string. Do not attempt to free the returned pointer. The **getenv** subroutine returns a pointer to the user's copy of the environment (which is static), until the first invocation of the **putenv** subroutine that adds a new environment variable. The **putenv** subroutine allocates an area of memory large enough to hold both the user's environment and the new variable. The next call to the **getenv** subroutine returns a pointer to this newly allocated space that is not static. Subsequent calls by the **putenv** subroutine use the **realloc** subroutine to make space for new variables. Unsuccessful completion returns a null pointer.

getevars Subroutine

Purpose

Gets environment of a process.

Library

Standard C library (**libc.a)**

Syntax

```
#include <procinfo.h>
#include <sys/types.h>

int getevars (processBuffer, bufferLen, argsBuffer, argsLen)
struct procsinfo  *processBuffer
or struct procentry64  *processBuffer;
int  bufferLen;
char  *argsBuffer;
int  argsLen;
```

Description

The **getevars** subroutine returns the environment that was passed to a command when it was started. Only one process can be examined per call to **getevars**.

The **getevars** subroutine uses the pi_pid field of *processBuffer* to determine which process to look for. *bufferLen* should be set to size of **struct procsinfo** or **struct procentry64**. Parameters are returned in *argsBuffer*, which should be allocated by the caller. The size of this array must be given in *argsLen*.

On return, *argsBuffer* consists of a succession of strings, each terminated with a null character (ascii `\0'). Hence, two consecutive NULLs indicate the end of the list.

Note: The arguments may be changed asynchronously by the process, but results are not guaranteed to be consistent.

Parameters

processBuffer
Specifies the address of a **procsinfo** or **procentry64** structure, whose pi_pid field should contain the pid of the process that is to be looked for.

bufferLen
Specifies the size of a single **procsinfo** or **procentry64** structure.

argsBuffer
Specifies the address of an array of characters to be filled with a series of strings representing the parameters that are needed. An extra NULL character marks the end of the list. This array must be allocated by the caller.

argsLen
Specifies the size of the *argsBuffer* array. No more than *argsLen* characters are returned.

Return Values

If successful, the **getevars** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getevars** subroutine does not succeed if the following are true:

Item	Description
ESRCH	The specified process does not exist.

Item	Description
EFAULT	The copy operation to the buffer was not successful or the *processBuffer* or *argsBuffer* parameters are invalid.
EINVAL	The *bufferLen* parameter does not contain the size of a single **procsinfo** or **procentry64** structure.
ENOMEM	There is no memory available in the address space.

getfilehdr Subroutine

Purpose

Retrieves the header details of the advanced accounting data file.

Library

The `libaacct.a` library.

Syntax

```
#define <sys/aacct.h>
getfilehdr(filename, hdrinfo)
char *filename;
struct aacct_file_header *hdrinfo;
```

Description

The `getfilehdr` subroutine retrieves the advanced accounting data file header information in a structure of type `aacct_file_header` and returns it to the caller through the structure pointer passed to it. The data file header contains the system details such as the name of the host, the partition number, and the system model.

Parameters

Item	Description
filename	Name of the advanced accounting data file.
hdrinfo	Pointer to the `aacct_file_header` structure in which the header information is returned.

Security

No restrictions. Any user can call this function.

Return Values

Item	Description
0	The call to the subroutine was successful.
-1	The call to the subroutine failed.

Error Codes

Item	Description
EINVAL	The passed pointer is NULL.

Item	Description
ENOENT	Specified data file does not exist.
EPERM	Permission denied. Unable to read the data file.

getfirstprojdb Subroutine

Purpose

Retrieves details of the first project from the specified project database.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>
getfirstprojdb(void *handle, struct project *project, char *comm)
```

Description

The **getfirstprojdb** subroutine retrieves the first project definitions from the project database, which is controlled through the *handle* parameter. The caller must initialize the project database prior to calling this routine with the **projdballoc** routine. Upon successful completion, the project information is copied to the project structure specified by the caller. In addition, the associated project comment, if present, is copied to the buffer pointed to by the *comm* parameter. The comment buffer is allocated by the caller and must have a length of 1024 bytes.

There is an internal state (that is, the current project) associated with the project database. When the project database is initialized, the current project is the first project in the database. The **getnextprojdb** subroutine returns the current project and advances the current project assignment to the next project in the database so that successive calls read each project entry in the database. The **getfirstprojdb** subroutine can be used to reset the database, so that the initial project is the current project assignment.

Parameters

Item	Description
handle	Pointer to the **projdb** handle.
project	Pointer to project structure where the retrieved data is stored.
comm	Pointer to the comment buffer.

Security

No restriction. Any user can call this function.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Invalid arguments, if passed pointer is NULL.
ENOENT	No projects available.

getfsent, getfsspec, getfsfile, getfstype, setfsent, or endfsent Subroutine

Purpose

Gets information about a file system.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <fstab.h>
```

```
struct fstab *getfsent( )
```

struct fstab *getfsspec (*Special*)
char *Special;

struct fstab *getfsfile(*File*)
char *File;

struct fstab *getfstype(*Type*)
char *Type;

```
void setfsent( )
```

```
void endfsent( )
```

Description

The **getfsent** subroutine reads the next line of the **/etc/filesystems** file, opening the file if necessary.

The **setfsent** subroutine opens the **/etc/filesystems** file and positions to the first record.

The **endfsent** subroutine closes the **/etc/filesystems** file.

The **getfsspec** and **getfsfile** subroutines sequentially search from the beginning of the file until a matching special file name or file-system file name is found, or until the end of the file is encountered. The **getfstype** subroutine does likewise, matching on the file-system type field.

Note: All information is contained in a static area, which must be copied to be saved.

Parameters

Item	Description
Special	Specifies the file-system file name.
File	Specifies the file name.

Item	Description
Type	Specifies the file-system type.

Return Values

The **getfsent**, **getfsspec**, **getfstype**, and **getfsfile** subroutines return a pointer to a structure that contains information about a file system. The header file **fstab.h** describes the structure. A null pointer is returned when the end of file (EOF) is reached or if an error occurs.

Files

Item	Description
/etc/filesystems	Centralizes file system characteristics.

getfsfbitindex and getfsfbitstring Subroutines

Purpose

Retrieve file security flag indices and strings.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>

int getfsfbitindex (fsfname)
const char *fsfname;

int getfsfstring (buff, size, index)
char *buff;
int *size;
int index;
```

Description

The **getfsfbitindex** subroutine searches in the file security flags table for the file security flag that the *fsfname* parameter specifies. The file security flag name that the *fsfname* parameter specified is used as a string.

The **getfsfstring** subroutine converts the specified file security flag index into a string and stores the result in the *buff* parameter. The length of the *buff* parameter is specified by the *size* parameter. If the length specified by the *size* parameter is less than that of the string, the **getfsfstring** subroutine fails and returns the required length into the *size* parameter for the index that is specified by the *index* parameter.

Parameters

Item	Description
buff	Specifies the buffer that the file security flag is copied to.
fsfname	Specifies the file security flag to be searched for.
index	Specifies the file security flag index that is to be converted to a string.
size	Specifies the size of the buffer that the *buff* parameter specifies.

Return Values

If successful, the **getfsfbitindex** subroutine returns a value that is equal to or greater than zero. Otherwise, it returns a value of -1.

If successful, the **getfsfstring** subroutine returns a value of zero. Otherwise, it returns a value of -1.

Error Codes

If these subroutines fail, they set one of the following error codes:

Item	Description
EINVAL	The parameters specified NULL value or the index is not valid.
ENOSPC	The size of the buffer is not large enough to store the file security flag string.

getgid, getegid or gegidx Subroutine

Purpose

Gets the process group IDs.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
#include <sys/types.h>
```

```
gid_t getgid (void);
```

```
gid_t getegid (void);
```

```
#include <id.h>
```

```
gid_t getgidx (int type);
```

Description

The **getgid** subroutine returns the real group ID of the calling process.

The **getegid** subroutine returns the effective group ID of the calling process.

The **getgidx** subroutine returns the group ID indicated by the *type* parameter of the calling process.

These subroutines are part of Base Operating System (BOS) Runtime.

Return Values

The **getgid**, **getegid**and **getgidx** subroutines return the requested group ID. The **getgid** and **getegid** subroutines are always successful.

The **getgidx** subroutine will return -1 and set the global **errno** variable to EINVAL if the type parameter is not one of ID_REAL, ID_EFFECTIVE or ID_SAVED.

Parameters

Item	Description
type	Specifies the group ID to get. Must be one of `ID_REAL` (real group ID), `ID_EFFECTIVE` (effective group ID) or `ID_SAVED` (saved set-group ID).

Error Codes

If the **getgidx** subroutine fails the following is returned:

Item	Description
`EINVAL`	Indicates the value of the type parameter is invalid.

getgrent, getgrgid, getgrnam, setgrent, or endgrent Subroutine

Purpose

Accesses the basic group information in the user database.

Library

Standard C Library (**libc.a)**

Syntax

```
#include <sys/types.h>
#include <grp.h>
```

```
struct group *getgrent ( );
```

```
struct group *getgrgid (GID)
gid_t  GID;
```

```
struct group *getgrnam (Name)
const char * Name;
```

```
void setgrent ( );
```

```
void endgrent ( );
```

Description

Attention: The information returned by the **getgrent**, **getgrnam**, and **getgrgid** subroutines is stored in a static area and is overwritten on subsequent calls. You must copy this information to save it.

Attention: These subroutines should not be used with the **getgroupattr** subroutine. The results are unpredictable.

The **setgrent** subroutine opens the user database if it is not already open. Then, this subroutine sets the cursor to point to the first group entry in the database.

Attention: The **getgrent** subroutine is only supported by LOCAL and NIS load modules, not any other LAM authentication module.

The **getgrent**, **getgrnam**, and **getgrgid** subroutines return information about the requested group. The **getgrent** subroutine returns the next group in the sequential search. The **getgrnam** subroutine returns the first group in the database whose name matches that of the *Name* parameter. The **getgrgid**

subroutine returns the first group in the database whose group ID matches the *GID* parameter. The **endgrent** subroutine closes the user database.

Note: An ! (exclamation mark) is written into the `gr_passwd` field. This field is ignored and is present only for compatibility with older versions of UNIX operating systems.

Note: If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the **getgrnam** or **getgrgid** subroutine gets group information from the Lightweight Directory Access Protocol (LDAP) and files domains, if the group name or group ID belongs to any one of these domains.

These subroutines also return the list of user members for the group. Currently, the list is limited to 2000 entries (this could change in the future to where all the entries for the group are returned).

The Group Structure

The **group** structure is defined in the **grp.h** file and has the following fields:

Item	Description
`gr_name`	Contains the name of the group.
`gr_passwd`	Contains the password of the group. **Note:** This field is no longer used.
`gr_gid`	Contains the ID of the group.
`gr_mem`	Contains the members of the group.

If the Network Information Service (NIS) is enabled on the system, these subroutines attempt to retrieve the group information from the NIS authentication server.

Parameters

Item	Description
GID	Specifies the group ID.
Name	Specifies the group name.

Item	Description
Group	Specifies the basic group information to enter into the user database.

Return Values

If successful, the **getgrent**, **getgrnam**, and **getgrgid** subroutines return a pointer to a valid group structure. Otherwise, a null pointer is returned.

Error Codes

These subroutines fail if one or more of the following are returned:

Item	Description
EIO	Indicates that an input/output (I/O) error has occurred.
EINTR	Indicates that a signal was caught during the **getgrnam** or **getgrgid** subroutine.
EMFILE	Indicates that the maximum number of file descriptors specified by the **OPEN_MAX** value are currently open in the calling process.
ENFILE	Indicates that the maximum allowable number of files is currently open in the system.

To check an application for error situations, set the **errno** global variable to a value of 0 before calling the **getgrgid** subroutine. If the **errno** global variable is set on return, an error occurred.

File

Item	Description
/etc/group	Contains basic group attributes.

getgrgid_r Subroutine

Purpose

Gets a group database entry for a group ID.

Library

Thread-safe C Library (**libc_r.a**)

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrgid_r(gid_t gid,
struct group *grp,
char *buffer,
size_t bufsize,
struct group **result);
```

Description

The **getgrgid_r** subroutine updates the **group** structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching *gid*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Note: If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the **getgrgid_r** subroutine gets group information from the Lightweight Directory Access Protocol and files domains, if the group ID belongs to any one of these domains.

Return Values

Upon successful completion, **getgrgid_r** returns a pointer to a **struct group** with the structure defined in **<grp.h>** with a matching entry if one is found. The **getgrgid_r** function returns a null pointer if either the requested entry was not found, or an error occurred. On error, *errno* will be set to indicate the error.

The return value points to a static area that is overwritten by a subsequent call to the **getgrent**, **getgrgid**, or **getgrnam** subroutine.

If successful, the **getgrgid_r** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **getgrgid_r** function fails if:

Item	Description
ERANGE	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **group** structure.

Applications wishing to check for error situations should set *errno* to 0 before calling **getgrgid_r**. If *errno* is set on return, an error occurred.

getgrnam_r Subroutine

Purpose

Search a group database for a name.

Library

Thread-Safe C Library (**libc_r.a**)

Syntax

```
#include <sys/types.h>
#include <grp.h>

int getgrnam_r (const char **name,
struct group *grp,
char *buffer,
size_t bufsize,
struct group **result);
```

Description

The **getgrnam_r** function updates the **group** structure pointed to by *grp* and stores pointer to that structure at the location pointed to by *result.* The structure contains an entry from the group database with a matching *gid* or *name*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. The maximum size needed for this buffer can be determined with the {_SC_GETGR_R_SIZE_MAX} *sysconf* parameter. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Note: If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file then the **getgrnam_r** subroutine gets group information from the Lightweight Directory Access Protocol (LDAP) and files, if the group name belongs to any one of these domains.

Return Values

The **getgrnam_r** function returns a pointer to a **struct group** with the structure defined in **<grp.h>** with a matching entry if one is found. The **getgrnam_r** function returns a null pointer if either the requested entry was not found, or an error occurred. On error, *errno* will be set to indicate the error.

The return value points to a static area that is overwritten by a subsequent call to the **getgrent**, **getgrgid**, or **getgrnam** subroutine.

If successful, the **getgrnam_r** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **getgrnam_r** function fails if:

Item	Description
ERANGE	Insufficient storage was supplied via *buffer* and *bufsize* to contain the data to be referenced by the resulting **group** structure.

Applications wishing to check for error situations should set *errno* to 0 before calling **getgrnam_r**. If *errno* is set on return, an error occurred.

getgroupattr, IDtogroup, nextgroup, or putgroupattr Subroutine

Purpose

Accesses the group information in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int getgroupattr (Group, Attribute, Value, Type)
char * Group;
char * Attribute;
void * Value;
int  Type;
```

```
int putgroupattr (Group, Attribute, Value, Type)
char *Group;
char *Attribute;
void *Value;
int Type;
```

```
char *IDtogroup ( GID)
gid_t GID;
```

```
char *nextgroup ( Mode,  Argument)
int Mode, Argument;
```

Description

Attention: These subroutines and the **setpwent** and **setgrent** subroutines should not be used simultaneously. The results can be unpredictable.

These subroutines access group information. Because of their greater granularity and extensibility, you should use them instead of the **getgrent**, **putgrent**, **getgrnam**, **getgrgid**, **setgrent**, and **endgrent** subroutines.

The **getgroupattr** subroutine reads a specified attribute from the group database. If the database is not already open, the subroutine will do an implicit open for reading.

Similarly, the **putgroupattr** subroutine writes a specified attribute into the group database. If the database is not already open, the subroutine does an implicit open for reading and writing. Data changed by **putgroupattr** must be explicitly committed by calling the **putgroupattr** subroutine with a *Type* parameter specifying the **SEC_COMMIT** value. Until the data is committed, only **get** subroutine calls within the process will return the written data.

New entries in the user and group databases must first be created by invoking **putgroupattr** with the **SEC_NEW** type.

The **IDtogroup** subroutine translates a group ID into a group name.

The **nextgroup** subroutine returns the next group in a linear search of the group database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user database.

Parameters

Item	Description
Argument	Presently unused and must be specified as null.
Attribute	Specifies which attribute is read. The following possible values are defined in the **usersec.h** file: **S_ID** Group ID. The attribute type is **SEC_INT**. **S_USERS** Members of the group. The attribute type is **SEC_LIST**. **S_ADMS** Administrators of the group. The attribute type is **SEC_LIST**. **S_ADMIN** Administrative status of a group. Type: **SEC_BOOL**. **S_GRPEXPORT** Specifies if the DCE registry can overwrite the local group information with the DCE group information during a DCE export operation. The attribute type is **SEC_BOOL**. Additional user-defined attributes may be used and will be stored in the format specified by the *Type* parameter.
GID	Specifies the group ID to be translated into a group name.
Group	Specifies the name of the group for which an attribute is to be read.
Mode	Specifies the search mode. Also can be used to delimit the search to one or more user credential databases. Specifying a non-null *Mode* value implicitly rewinds the search. A null mode continues the search sequentially through the database. This parameter specifies one of the following values as a bit mask (defined in the **usersec.h** file): **S_LOCAL** The local database of groups are included in the search. **S_SYSTEM** All credentials servers for the system are searched.

Item	Description
Type	Specifies the type of attribute expected. Valid values are defined in the **usersec.h** file and include: **SEC_INT** The format of the attribute is an integer. The buffer returned by the **getgroupattr** subroutine and the buffer supplied by the **putgroupattr** subroutine are defined to contain an integer. **SEC_CHAR** The format of the attribute is a null-terminated character string. **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters. **SEC_BOOL** A pointer to an integer (**int ***) that has been cast to a null pointer. **SEC_COMMIT** For the **putgroupattr** subroutine, this value specified by itself indicates that changes to the named group are committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no group is specified, changes to all modified groups are committed to permanent storage. **SEC_DELETE** The corresponding attribute is deleted from the database. **SEC_NEW** If using the **putgroupattr** subroutine, updates all the group database files with the new group name.
Value	Specifies the address of a pointer for the **getgroupattr** subroutine. The **getgroupattr** subroutine will return the address of a buffer in the pointer. For the **putgroupattr** subroutine, the *Value* parameter specifies the address of a buffer in which the attribute is stored. See the *Type* parameter for more details.

Security

Item	Description
Files Accessed:	

Mode	File
rw	**/etc/group** (write access for **putgroupattr**)
rw	**/etc/security/group** (write access for **putgroupattr**)

Return Values

The **getgroupattr** and **putgroupattr** subroutines, when successfully completed, return a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

The **IDtogroup** and **nextgroup** subroutines return a character pointer to a buffer containing the requested group name, if successfully completed. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

Note: All of these subroutines return errors from other subroutines.

These subroutines fail if the following is true:

Item	Description
EACCES	Access permission is denied for the data request.

The **getgroupattr** subroutine fails if the following is returned:

Item	Description
EIO	Failed to access remote group database.

The **getgroupattr** and **putgroupattr** subroutines fail if one or more of the following are true:

Item	Description
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors may not be detected.
EINVAL	The *Type* parameter contains more than one of the **SEC_INT**, **SEC_BOOL**, **SEC_CHAR**, **SEC_LIST**, or **SEC_COMMIT** attributes.
EINVAL	The *Type* parameter specifies that an individual attribute is to be committed, and the *Group* parameter is null.
ENOENT	The specified *Group* parameter does not exist or the attribute is not defined for this group.
EPERM	Operation is not permitted.

The **IDtogroup** subroutine fails if the following is true:

Item	Description
ENOENT	The *GID* parameter could not be translated into a valid group name on the system.

The **nextgroup** subroutine fails if one or more of the following are true:

Item	Description
EINVAL	The *Mode* parameter is not null, and does not specify either **S_LOCAL** or **S_SYSTEM**.
EINVAL	The *Argument* parameter is not null.
ENOENT	The end of the search was reached.

getgroupattrs Subroutine

Purpose

Retrieves multiple group attributes in the group database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getgroupattrs (Group, Attributes, Count)
char * Group;
```

```
dbattr_t * Attributes;
int Count
```

Description

Attention: Do not use this subroutine and the **setpwent** and **setgrent** subroutines simultaneously. The results can be unpredictable.

The **getgroupattrs** subroutine accesses group information. Because of its greater granularity and extensibility, use it instead of the **getgrent** routines.

The **getgroupattrs** subroutine reads one or more attributes from the group database. If the database is not already open, this subroutine does an implicit open for reading. A call to the **getgroupattrs** subroutine with an *Attributes* parameter of null and *Count* parameter of 0 for every new group verifies that the group exists.

The *Attributes* array contains information about each attribute that is to be read. The **dbattr_t** data structure contains the following fields:

attr_name
The name of the desired attribute.

attr_idx
Used internally by the **getgroupattrs** subroutine.

attr_type
The type of the desired attribute. The list of attribute types is defined in the **usersec.h** header file.

attr_flag
The results of the request to read the desired attribute.

attr_un
A union containing the returned values. Its union members that follow correspond to the definitions of the **attr_char**, **attr_int**, **attr_long**, and **attr_llong** macros, respectively:

au_char
Attributes of type **SEC_CHAR** and **SEC_LIST** store a pointer to the returned attribute in this member when the requested attribute is successfully read. The caller is responsible for freeing this memory.

au_int
Attributes of type **SEC_INT** and **SEC_BOOL** store the value of the attribute in this member when the requested attribute is successfully read.

au_long
Attributes of type **SEC_LONG** store the value of the attribute in this member when the requested attribute is successfully read.

au_llong
Attributes of type **SEC_LLONG** store the value of the attribute in this member when the requested attribute is successfully read.

attr_domain
The authentication domain containing the attribute. The **getgroupattrs** subroutine is responsible for managing the memory referenced by this pointer. If **attr_domain** is specified for an attribute, the get request is sent only to that domain. If **attr_domain** is not specified (that is, set to NULL), **getgroupattrs** searches the domains in a predetermined order. The search starts with the local file system and continues with the domains specified in the **/usr/lib/security/methods.cfg** file. This search space can be restricted with the **setauthdb** subroutine so that only the domain specified in the **setauthdb** call is searched. If **attr_domain** is not specified, the **getgroupattrs** subroutine sets this field to the name of the domain from which the value is retrieved. If the request for a NULL domain was not satisfied, the request is tried from the local files using the default stanza.

Use the **setuserdb** and **enduserdb** subroutines to open and close the group database. Failure to explicitly open and close the group database can result in loss of memory and performance.

Parameters

Item	Description
Group	Specifies the name of the group for which the attributes are to be read.
Attributes	A pointer to an array of 0 or more elements of type **dbattr_t**. The list of group attributes is defined in the **usersec.h** header file.
Count	The number of array elements in *Attributes*. A *Count* parameter of 0 can be used to determine if the group exists.

Security

Files accessed:

Item	Description
Mode	**File**
rw	**/etc/group**
rw	**/etc/security/group**

Return Values

If *Group* exists, the **getgroupattrs** subroutine returns 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error. Each element in the *Attributes* array must be examined on a successful call to **getgroupattrs** to determine if the *Attributes* array entry was successfully retrieved.

Error Codes

The **getgroupattrs** subroutine returns an error that indicates that the group does or does not exist. Additional errors can indicate an error querying the information databases for the requested attributes.

Item	Description
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Attributes* parameter is null and the *Count* parameter is greater than 0.
ENOENT	The specified *Group* parameter does not exist.
EIO	Failed to access the remote group database.

If the **getgroupattrs** subroutine fails to query an attribute, one or more of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The user does not have access to the attribute specified in the *attr_name* field.
EINVAL	The **attr_type** field in the *Attributes* entry contains an invalid type.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
ENOATTR	The **attr_name** field in the *Attributes* entry specifies an attribute that is not defined for this user or group.
ENOMEM	Memory could not be allocated to store the return value.

Examples

The following sample test program displays the output to a call to **getgroupattrs**. In this example, the system has a user named foo.

```
    attribute name   : id
    attribute flag   : 0
    attribute domain : files
    attribute value  : 204

    attribute name   : admin
    attribute flag   : 0
    attribute domain : files
    attribute value  : 0

    attribute name   : adms
    attribute flag   : 0
    attribute domain : files
    attribute value  :

    attribute name   : registry
    attribute flag   : 0
    attribute domain :
    attribute value  : compat

 */
#include <stdio.h>
#include <usersec.h>

#define NATTR 4
#define GROUPNAME "bar"

char * ConvertToComma(char *);  /* Convert from SEC_LIST to SEC_CHAR with
                   '\0' replaced with ',' */
main() {

    dbattr_t attributes[NATTR];
    int     i;
    int    rc;

    memset (&attributes, 0, sizeof(attributes));

    /*
     * Fill in the attributes array with "id", "expires" and
     * "SYSTEM" attributes.
     */

    attributes[0].attr_name = S_ID;
    attributes[0].attr_type = SEC_INT;;

    attributes[1].attr_name = S_ADMIN;
    attributes[1].attr_type = SEC_BOOL;

    attributes[2].attr_name = S_ADMS;
    attributes[2].attr_type = SEC_LIST;

    attributes[3].attr_name = S_REGISTRY;
    attributes[3].attr_type = SEC_CHAR;

    /*
     * Make a call to getuserattrs.
     */
        setuserdb(S_READ);

    rc = getgroupattrs(GROUPNAME, attributes, NATTR);

        enduserdb();

    if (rc) {
        printf("getgroupattrsattrs failed ....\n");
        exit(-1);
    }

    for (i = 0; i < NATTR; i++) {
        printf("attribute name   : %s \n", attributes[i].attr_name);
        printf("attribute flag   : %d \n", attributes[i].attr_flag);

        if (attributes[i].attr_flag) {

            /*
```

```
             * No attribute value. Continue.
             */
            printf("\n");
            continue;
        }
        /*
         * We have a value.
         */
        printf("attribute domain : %s \n", attributes[i].attr_domain);
        printf("attribute value  : ");

        switch (attributes[i].attr_type)
        {
            case SEC_CHAR:
                if (attributes[i].attr_char) {
                                printf("%s\n", attributes[i].attr_char);
                    free(attributes[i].attr_char);
                }
                break;
            case SEC_LIST:
                if (attributes[i].attr_char) {
                                printf("%s\n", ConvertToComma(
                            attributes[i].attr_char));
                    free(attributes[i].attr_char);
                }
                                break;
            case SEC_INT:
            case SEC_BOOL:
                                printf("%d\n", attributes[i].attr_int);
                                break;
            default:
                break;
                }
        printf("\n");
    }
    exit(0);
}

/*
 * ConvertToComme:
 * replaces NULLs in str with commas.
 */
char *
ConvertToComma(char *str)
{
        char *s = str;

        if (! str || ! *str)
                return(s);

        for (; *str; str++) {
                while(*(++str));
                *str = ',';
        }

        *(str-1) = 0;
        return(s);
}
```

The following output for the call is expected:

```
    attribute name   : id
    attribute flag   : 0
    attribute domain : files
    attribute value  : 204

    attribute name   : admin
    attribute flag   : 0
    attribute domain : files
    attribute value  : 0

    attribute name   : adms
    attribute flag   : 0
    attribute domain : files
    attribute value  :

    attribute name   : registry
    attribute flag   : 0
    attribute domain :
    attribute value  : compat
```

Files

Item	Description
/etc/group	Contains group IDs.

getgroups Subroutine

Purpose

Gets the supplementary group ID of the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <unistd.h>

int getgroups (NGroups, GIDSet)
int  NGroups;
gid_t  GIDSet [ ];
```

Description

The **getgroups** subroutine gets the supplementary group ID of the process. The list is stored in the array pointed to by the *GIDSet* parameter. The *NGroups* parameter indicates the number of entries that can be stored in this array. The **getgroups** subroutine never returns more than the number of entries specified by the **NGROUPS_MAX** constant. (The **NGROUPS_MAX** constant is defined in the **limits.h** file.) If the value in the *NGroups* parameter is 0, the **getgroups** subroutine returns the number of groups in the supplementary group.

Parameters

Item	Description
GIDSet	Points to the array in which the supplementary group ID of the user's process is stored.
NGroups	Indicates the number of entries that can be stored in the array pointed to by the *GIDSet* parameter.

Return Values

Upon successful completion, the **getgroups** subroutine returns the number of elements stored into the array pointed to by the *GIDSet* parameter. If the **getgroups** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getgroups** subroutine is unsuccessful if either of the following error codes is true:

Item	Description
EFAULT	The *NGroups* and *GIDSet* parameters specify an array that is partially or completely outside of the allocated address space of the process.

Item	Description
EINVAL	The *NGroups* parameter is smaller than the number of groups in the supplementary group.

getgrpaclattr, nextgrpacl, or putgrpaclattr Subroutine

Purpose

Accesses the group screen information in the SMIT ACL database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int getgrpaclattr (Group, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

```
char *nextgrpacl(void)
```

```
int putgrpaclattr (Group, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description

The **getgrpaclattr** subroutine reads a specified group attribute from the SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the **putgrpaclattr** subroutine writes a specified attribute into the user SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putgrpaclattr** subroutine must be explicitly committed by calling the **putgrpaclattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getgrpaclattr** subroutine within the process returns written data.

The **nextgrpacl** subroutine returns the next group in a linear search of the group SMIT ACL database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setacldb** and **endacldb** subroutines should be used to open and close the database.

Parameters

Item	Description
Attribute	Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file: **S_SCREENS** String of SMIT screens. The attribute type is **SEC_LIST**.

Item	Description
Type	Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include: **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string. For the **getgrpaclattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putgrpaclattr** subroutine, the user should supply a character pointer. **SEC_COMMIT** For the **putgrpaclattr** subroutine, this value specified by itself indicates that changes to the named group are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no group is specified, the changes to all modified groups are committed to permanent storage. **SEC_DELETE** The corresponding attribute is deleted from the group SMIT ACL database. **SEC_NEW** Updates the group SMIT ACL database file with the new group name when using the **putgrpaclattr** subroutine.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Return Values

If successful, the **getgrpaclattr** returns 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Possible return codes are:

Item	Description
EACCES	Access permission is denied for the data request.
ENOENT	The specified *Group* parameter does not exist or the attribute is not defined for this group.
ENOATTR	The specified user attribute does not exist for this group.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or null.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
EPERM	Operation is not permitted.

getgrset Subroutine

Purpose

Accesses the concurrent group set information in the user database.

Library

Standard C Library (**libc.a)**

Syntax

```
char *getgrset (User)
const char * User;
```

Description

The **getgrset** subroutine returns a pointer to the comma separated list of concurrent group identifiers for the named user.

If the Network Information Service (NIS) is enabled on the system, these subroutines attempt to retrieve the user information from the NIS authentication server.

Notes:

- If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, all the group IDs are fetched from the Lightweight Directory Access Protocol (LDAP) and the files domains, if the user belongs to any one of these domains.
- The **getgrset** subroutine is not a threadsafe subroutine. For information about the threadsafe subroutine, see the **getgrset_r** subroutine.

Parameters

Item	Description
User	Specifies the user name.

Return Values

If successful, the **getgrset** subroutine returns a pointer to a list of supplementary groups. This pointer must be freed by the user.

Error Codes

A **NULL** pointer is returned on error. The value of the **errno** global variable is undefined on error.

File

Item	Description
/etc/group	Contains basic group attributes.

getgrset_r Subroutine

Purpose

Obtains group set information in the user database.

Library

Threadsafe C Library (**libc.a)**

Syntax

#include <sys/types.h>

#include <grp.h>

#define _THREAD_SAFE

int **getgrset_r** (char *nam, struct _grjunk * grp)

Description

The **getgrset_r** subroutine populates group data into the structure pointed by group for the named user. It returns TS_SUCCESS if the group information is populated.

If the Network Information Service (NIS) is enabled on the system, these subroutines attempt to retrieve the user information from the NIS authentication server.

Note: If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, all the group IDs are fetched from the Lightweight Directory Access Protocol (LDAP) and files domains, if the user belongs to any one of these domains.

Parameters

Item	Description
User	Specifies the user name.

Return Values

If successful, the **getgrset_r** subroutine fills the group-related information into the second parameter and returns 0.

Error Codes

The **getgrset_r** subroutine returns -1 when it is unable to populate the **struct_grjunk *grp** structure.

File

Item	Description
/etc/group	Contains basic group attributes.

getinterval, incinterval, absinterval, resinc, resabs, alarm, ualarm, getitimer or setitimer Subroutine

Purpose

Manipulates the expiration time of interval timers.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>

int getinterval ( timerID,  value)
timer_t timerID;
struct itimerstruc_t *value;

int incinterval (timerID, value,  ovalue)
timer_t timerID;
struct itimerstruc_t *value, *ovalue;
```

```
int absinterval (timerID, value, ovalue)
timer_t timerID;
struct itimerstruc_t *value, *ovalue;

int resabs (timerID,  resolution,  maximum)
timer_t timerID;
struct timestruc_t *resolution, *maximum;

int resinc (timerID, resolution, maximum)
timer_t timerID;
struct timestruc_t *resolution, *maximum;
```

```
#include <unistd.h>
```

```
unsigned int alarm ( seconds)
unsigned int seconds;

useconds_t ualarm (value,  interval)
useconds_t value, interval;

int setitimer ( which, value, ovalue)
int which;
struct itimerval *value, *ovalue;

int getitimer (which, value)
int which;
struct itimerval *value;
```

Description

The **getinterval**, **incinterval**, and **absinterval** subroutines manipulate the expiration time of interval timers. These functions use a timer value defined by the **struct itimerstruc_t** structure, which includes the following fields:

```
struct timestruc_t it_interval;   /* timer interval period      */
struct timestruc_t it_value;      /* timer interval expiration  */
```

If the `it_value` field is nonzero, it indicates the time to the next timer expiration. If `it_value` is 0, the per-process timer is disabled. If the `it_interval` member is nonzero, it specifies a value to be used in reloading the `it_value` field when the timer expires. If `it_interval` is 0, the timer is to be disabled after its next expiration (assuming `it_value` is nonzero).

The **getinterval** subroutine returns a value from the **struct itimerstruc_t** structure to the *value* parameter. The `it_value` field of this structure represents the amount of time in the current interval before the timer expires, should one exist for the per-process timer specified in the *timerID* parameter. The `it_interval` field has the value last set by the **incinterval** or **absinterval** subroutine. The fields of the *value* parameter are subject to the resolution of the timer.

The **incinterval** subroutine sets the value of a per-process timer to a given offset from the current timer setting. The **absinterval** subroutine sets the value of the per-process timer to a given absolute value. If the specified absolute time has already expired, the **absinterval** subroutine will succeed and the expiration notification will be made. Both subroutines update the interval timer period. Time values smaller than the resolution of the specified timer are rounded up to this resolution. Time values larger than the maximum value of the specified timer are rounded down to the maximum value.

The **resinc** and **resabs** subroutines return the resolution and maximum value of the interval timer contained in the *timerID* parameter. The resolution of the interval timer is contained in the *resolution* parameter, and the maximum value is contained in the *maximum* parameter. These values might not be

the same as the values returned by the corresponding system timer, the **gettimer** subroutine. In addition, it is likely that the maximum values returned by the **resinc** and **resabs** subroutines will be different.

Note: If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request is raised to 10 milliseconds.

The **alarm** subroutine causes the system to send the calling thread's process a **SIGALRM** signal after the number of real-time seconds specified by the *seconds* parameter have elapsed. Since the signal is sent to the process, in a multi-threaded process another thread than the one that called the **alarm** subroutine may receive the **SIGALRM** signal. Processor scheduling delays may prevent the process from handling the signal as soon as it is generated. If the value of the *seconds* parameter is 0, a pending alarm request, if any, is canceled. Alarm requests are not stacked. Only one **SIGALRM** generation can be scheduled in this manner. If the **SIGALRM** signal has not yet been generated, the call results in rescheduling the time at which the **SIGALRM** signal is generated. If several threads in a process call the **alarm** subroutine, only the last call will be effective.

The **ualarm** subroutine sends a **SIGALRM** signal to the invoking process in a specified number of seconds. The **getitimer** subroutine gets the value of an interval timer. The **setitimer** subroutine sets the value of an interval timer.

Parameters

Item	Description
timerID	Specifies the ID of the interval timer.
value	Points to a **struct itimerstruc_t** structure.
ovalue	Represents the previous time-out period.
resolution	Resolution of the timer.
maximum	Indicates the maximum value of the interval timer.
seconds	Specifies the number of real-time seconds to elapse before the first **SIGALRM** signal.
interval	Specifies the number of microseconds between subsequent periodic **SIGALRM** signals. If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request interval is automatically raised to 10 milliseconds.
which	Identifies the type of timer. Valid values are: **ITIMER_PROF** Decrements in process virtual time and when the system runs on behalf of the process. It is designed for use by interpreters in statistically profiling the execution of interpreted programs. Each time the **ITIMER_PROF** timer expires, the **SIGPROF** signal occurs. Because this signal may interrupt in-progress system calls, programs using this timer must be prepared to restart interrupted system calls. **ITIMER_REAL** Decrements in real time. A **SIGALRM** signal occurs when this timer expires. **ITMER_REAL_TH** Real-time, per-thread timer. Decrements in real time and delivers a **SIGTALRM** signal when it expires. The **SIGTALRM** is sent to the thread that sets the timer. Each thread has its own timer and can manipulate its own timer. This timer is only supported with the 1:1 thread model. If the timer is used in M:N thread model, undefined results might occur. **ITIMER_VIRTUAL** Decrements in process virtual time. It runs only during process execution. A **SIGVTALRM** signal occurs when it expires.

Return Values

If these subroutines are successful, a value of 0 is returned. If an error occurs, a value of -1 is returned and the **errno** global variable is set.

The **alarm** subroutine returns the amount of time (in seconds) remaining before the system is scheduled to generate the **SIGALARM** signal from the previous call to **alarm**. It returns a 0 if there was no previous **alarm** request.

The **ualarm** subroutine returns the number of microseconds previously remaining in the alarm clock.

Error Codes

If the **getinterval**, **incinterval**, **absinterval**, **resinc**, **resabs**, **setitimer**, **getitimer**, or **setitimer** subroutine is unsuccessful , a value of -1 is returned and the **errno** global variable is set to one of the following error codes:

Item	Description
EINVAL	Indicates that the *timerID* parameter does not correspond to an ID returned by the **gettimerid** subroutine, or a value structure specified a nanosecond value less than 0 or greater than or equal to one thousand million (1,000,000,000).
EIO	Indicates that an error occurred while accessing the timer device.
EFAULT	Indicates that a parameter address has referenced invalid memory.

The **alarm** subroutine is always successful. No return value is reserved to indicate an error for it.

getiopri Subroutine

Purpose

Enables the getting of a process I/O priority.

Syntax

```
short getiopri (ProcessID);
pid_t ProcessID;
```

Description

The `getiopri` subroutine returns the I/O scheduling priority of a process. If the target process ID does not match the process ID of the caller, the caller must either be running as root, or have an effective and real user ID that matches the target process.

Parameters

Item	Description
ProcessID	Specifies the process ID. If this value is `-1`, the current process scheduling priority is returned.

Return Values

Upon successful completion, the `getiopri` subroutine returns the I/O scheduling priority of a thread in the process. A returned value of `IOPRIORITY_UNSET` indicates that the I/O priority was not set. Otherwise, a value of `-1` is returned and the `errno` global variable is set to indicate the error.

Errors

Item	Description
EPERM	The calling process is not root. It does not have the same process ID as the target process, and does not have the same real effective user ID as the target process.
ESRCH	No process can be found corresponding to the specified *ProcessID*.

Implementation Specifics

1. Implementation requires an additional field in the `proc` structure.
2. The default setting for process I/O priority is `IOPRIORITY_UNSET`.
3. Once set, process I/O priorities should be inherited across a `fork`. I/O priorities should not be inherited across an `exec`.
4. The `setiopri` system call generates an auditing event using *audit_svcstart* if auditing is enabled on the system (*audit_flag* is true).

getipnodebyaddr Subroutine

Purpose

Address-to-nodename translation.

Library

Standard C Library (**libc.a**)

(**libaixinet**)

Syntax

```
#include <sys/socket.h>
#include <netdb.h>
struct hostent *getipnodebyaddr(src, len, af, error_num)
const void *src;
size_t len;
int af;
int *error_num;
```

Description

The **getipnodebyaddr** subroutine has the same arguments as the **gethostbyaddr** subroutine but adds an error number. It is thread-safe.

The **getipnodebyaddr** subroutine is similar in its name query to the **gethostbyaddr** subroutine except in one case. If *af* equals AF_INET6 and the IPv6 address is an IPv4-mapped IPv6 address or an IPv4-compatible address, then the first 12 bytes are skipped over and the last 4 bytes are used as an IPv4 address with af equal to AF_INET to lookup the name.

If the **getipnodebyaddr** subroutine is returning success, then the single address that is returned in the **hostent** structure is a copy of the first argument to the function with the same address family and length that was passed as arguments to this function.

All of the information returned by **getipnodebyaddr** is dynamically allocated: the **hostent** structure and the data areas pointed to by the `h_name`, `h_addr_lisy`, and `h_aliases` members of the **hostent** structure. To return this information to the system the function **freehostent** is called.

Parameters

Item	Description
src	Specifies a node address. It is a pointer to either a 4-byte (IPv4) or 16-byte (IPv6) binary format address.
af	Specifies the address family which is either AF_INET or AF_INET6.
len	Specifies the length of the node binary format address.
error_num	Returns argument to the caller with the appropriate error code.

Return Values

The **getipnodebyaddr** subroutine returns a pointer to a **hostent** structure on success.

The **getipnodebyaddr** subroutine returns a null pointer if an error occurs. The *error_num* parameter is set to indicate the error.

Error Codes

Item	Description
HOST_NOT_FOUND	The host specified by the *name* parameter was not found.
TRY_AGAIN	The local server did not receive a response from an authoritative server. Try again later.
NO_RECOVERY	This error code indicates an unrecoverable error.
NO_ADDRESS	The requested *name* is valid but does not have an Internet address at the name server.

getipnodebyname Subroutine

Purpose

Nodename-to-address translation.

Library

Standard C Library (**libc.a**)

(**libaixinet**)

Syntax

```
#include <libc.a>
#include <netdb.h>
struct hostent *getipnodebyname(name, af, flags, error_num)
const char *name;
int af;
int flags;
int *error_num;
```

Description

The commonly used functions **gethostbyname** and **gethostbyname2** are inadequate for many applications. You could not specify the type of addresses desired in **gethostbyname**. In **gethostbyname2**, a global option (RES_USE_INET6) is required when IPV6 addresses are used. Also, **gethostbyname2** needed more control over the type of addresses required.

The **getipnodebyname** subroutine gives the caller more control over the types of addresses required and is thread safe. It also does not need a global option like RES_USE_INET6.

The name argument can be either a node name or a numeric (either a dotted-decimal IPv4 or colon-seperated IPv6) address.

The *flags* parameter values include AI_DEFAULT, AI_V4MAPPED, AI_ALL and AI_ADDRCONFIG. The special flags value AI_DEFAULT is designed to handle most applications. Its definition is:

```
#define AI_DEFAULT (AI_V4MAPPED | AI_ADDRCONFIG)
```

When porting simple applications to use IPv6, simply replace the call:

```
hp = gethostbyname(name);
```

with

```
hp = getipnodebyname(name, AF_INET6, AI_DEFAULT, &error_num);
```

To modify the behavior of the **getipnodebyname** subroutine, constant values can be logically-ORed into the *flags* parameter.

A *flags* value of 0 implies a strict interpretation of the *af* parameter. If *af* is AF_INET then only IPv4 addresses are searched for and returned. If *af* is AF_INET6 then only IPv6 addresses are searched for and returned.

If the AI_V4MAPPED flag is specified along with an *af* of AF_INET6, then the caller accepts IPv4-mapped IPv6 addresses. That is, if a query for IPv6 addresses fails, then a query for IPv4 addresses is made and if any are found, then they are returned as IPv4-mapped IPv6 addresses. The AI_V4MAPPED flag is only valid with an *af* of AF_INET6.

If the AI_ALL flag is used in conjunction the AI_V4MAPPED flag and *af* is AF_INET6, then the caller wants all addresses. The addresses returned are IPv6 addresses and/or IPv4-mapped IPv6 addresses. Only if both queries (IPv6 and IPv4) fail does **getipnodebyname** return NULL. Again, the AI_ALL flag is only valid with an *af* of AF_INET6.

The AI_ADDRCONFIG flag is used to specify that a query for IPv6 addresses should only occur if the node has at least one IPv6 source address configured and a query for IPv4 addresses should only occur if the node has at least one IPv4 source address configured. For example, if the node only has IPv4 addresses configured, af equals AF_INET6, and the node name being looked up has both IPv4 and IPv6 addresses, then if only the AI_ADDRCONFIG flag is specified, getipnodebyname will return NULL. If the AI_V4MAPPED flag is specified with the AI_ADDRCONFIG flag (AI_DEFAULT), then any IPv4 addresses found will be returned as IPv4-mapped IPv6 addresses.

There are 4 different situations when the name argument is a literal address string:

1. *name* is a dotted-decimal IPv4 address and *af* is AF_INET. If the query is successful, then `h_name` points to a copy of *name*, `h_addrtype` is the *af* argument, `h_length` is 4, `h_aliases` is a NULL pointer, `h_addr_list[0]` points to the 4-byte binary address and `h_addr_list[1]` is a NULL pointer.
2. *name* is a colon-separated IPv6 address and *af* is AF_INET6. If the query is successful, then `h_name` points to a copy of *name*, `h_addrtype` is the *af* parameter, `h_length` is 16, `h_aliases` is a NULL pointer, `h_addr_list[0]` points to the 16-byte binary address and `h_addr_list[1]` is a NULL pointer.
3. *name* is a dotted-decimal IPv4 address and *af* is AF_INET6. If the AI_V4MAPPED flag is specified and the query is successful, then `h_name` points to an IPv4-mapped IPv6 address string, `h_addrtype` is the *af* argument, `h_length` is 16, `h_aliases` is a NULL pointer, `h_addr_list[0]` points to the 16-byte binary address and `h_addr_list[1]` is a NULL pointer.
4. *name* is a colon-separated IPv6 address and af is AF_INET. This is an error, **getipnodebyname** returns a NULL pointer and *error_num* equals HOST_NOT_FOUND.

Parameters

Item	Description
name	Specifies either a node name or a numeric (either a dotted-decimal IPv4 or colon-separated IPv6) address.
af	Specifies the address family which is either AF_INET or AF_INET6.
flags	Controls the types of addresses searched for and the types of addresses returned.
error_num	Returns argument to the caller with the appropriate error code.

Return Values

The **getipnodebyname** subroutine returns a pointer to a **hostent** structure on success.

The **getipnodebyname** subroutine returns a null pointer if an error occurs. The *error_num* parameter is set to indicate the error.

Error Codes

Item	Description
HOST_NOT_FOUND	The host specified by the *name* parameter was not found.
TRY_AGAIN	The local server did not receive a response from an authoritative server. Try again later.
NO_RECOVERY	The host specified by the *name*parameter was not found. This error code indicates an unrecoverable error.
NO_ADDRESS	The requested *name* is valid but does not have an Internet address at the name server.

getline, getdelim Subroutines

Purpose

Reads a delimited record from a stream.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
ssize_t getdelim(char **lineptr, size_t *n, int delimiter, FILE *stream);
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Description

The **getdelim** function reads from stream until it encounters a character matching the delimiter character. The delimiter argument is an int, the value of which the application will ensure is a character representable as an unsigned char of equal value that terminates the read process. If the delimiter argument has any other value, the behavior is undefined.

The application will ensure that *lineptr is a valid argument that could be passed to the free() function. If *n is non-zero, the application shall ensure that *lineptr points to an object of at least *n bytes.

The **getline()** function is equivalent to the **getdelim()** function with delimiter character equal to the '\n' character.

Return Values

Upon successful completion, the **getdelim()** function will return the number of characters written into the buffer, including the delimiter character if one was encountered before EOF. Otherwise, it returns -1 and set the errno to indicate the error.

Error Codes

The function may fail if:

Item	Description
[EINVAL]	lineptr or n are null pointers
[ENOMEM]	Insufficient memory is available.
[EINVAL]	Stream is not a valid file descriptor.
[EOVERFLOW]	More than {SSIZE_MAX} characters were read without encountering the delimiter character.

getlogin Subroutine

Purpose

Gets a user's login name.

Library

Standard C Library (**libc.a**)

Syntax

```
include <sys/types.h>
include <unistd.h>
include <limits.h>
```

```
char *getlogin (void)
```

Description

Attention: Do not use the **getlogin** subroutine in a multithreaded environment. To access the thread-safe version of this subroutines, see the **getlogin_r** (“getlogin_r Subroutine” on page 482) subroutine.

Attention: The **getlogin** subroutine returns a pointer to an area that may be overwritten by successive calls.

The **getlogin** subroutine returns a pointer to the login name in the /**etc**/**utmp** file. You can use the **getlogin** subroutine with the **getpwnam** (“getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine” on page 524) subroutine to locate the correct password file entry when the same user ID is shared by several login names.

If the **getlogin** subroutine cannot find the login name in the /**etc**/**utmp** file, it returns the process **LOGNAME** environment variable. If the **getlogin** subroutine is called within a process that is not attached

to a terminal, it returns the value of the **LOGNAME** environment variable. If the **LOGNAME** environment variable does not exist, a null pointer is returned.

In UNIX03 mode, if the login name cannot be found in the `/etc/utmp` file or if there is no controlling terminal for the process, the **getlogin** subroutine does not return the **LOGNAME** environment variable, it returns a null pointer and sets the error code **ENXIO**. This behavior is enabled by setting the environment variable **XPG_SUS_ENV=ON** (which enables all UNIX03 functionality) or by setting the variable **XPG_GETLOGIN=ON** (which just enables UNIX03 mode for the **getlogin** and **getlogin_r** subroutines).

Return Values

The return value can point to static data whose content is overwritten by each call. If the login name is not found, the **getlogin** subroutine returns a null pointer.

Error Codes

If the **getlogin** function is unsuccessful, it returns one or more of the following error codes:

Item	Description
EMFILE	Indicates that the **OPEN_MAX** file descriptors are currently open in the calling process.
ENFILE	Indicates that the maximum allowable number of files is currently open in the system.
ENXIO	Indicates that the calling process has no controlling terminal.

Files

Item	Description
/etc/utmp	Contains a record of users logged into the system.

getlogin_r Subroutine

Purpose

Gets a user's login name.

Library

Thread-Safe C Library (**libc_r.a**)

Syntax

```
int getlogin_r (Name, Length)
char * Name;
size_t  Length;
```

Description

The **getlogin_r** subroutine gets a user's login name from the **/etc/utmp** file and places it in the *Name* parameter. Only the number of bytes specified by the *Length* parameter (including the ending null value) are placed in the *Name* parameter.

Applications that call the **getlogin_r** subroutine must allocate memory for the login name before calling the subroutine. The name buffer must be the length of the *Name* parameter plus an ending null value.

If the **getlogin_r** subroutine cannot find the login name in the **utmp** file or the process is not attached to a terminal, it places the **LOGNAME** environment variable in the name buffer. If the **LOGNAME** environment variable does not exist, the *Name* parameter is set to null and the **getlogin_r** subroutine returns a -1.

In UNIX03 mode, if the login name cannot be found in the `/etc/utmp` file or if there is no controlling terminal for the process, the **getlogin_r** subroutine does not place the **LOGNAME** environment variable in the name buffer, it just returns the error code **ENXIO**. This behavior is enabled by setting the environment variable **XPG_SUS_ENV=ON** (which enables all UNIX03 functionality) or by setting the variable **XPG_GETLOGIN=ON** (which just enables UNIX03 mode for the **getlogin** and **getlogin_r** subroutines).

Parameters

Item	Description
Name	Specifies a buffer for the login name. This buffer should be the length of the *Length* parameter plus an ending null value.
Length	Specifies the total length in bytes of the *Name* parameter. No more bytes than the number specified by the *Length* parameter are placed in the *Name* parameter, including the ending null value.

Return Values

If successful, the `getlogin_r` function returns `0`. Otherwise, an error number is returned to indicate the error.

Error Codes

If the **getlogin_r** subroutine does not succeed, it returns one of the following error codes:

Item	Description
EINVAL	Indicates that the *Name* parameter is not valid.
EMFILE	Indicates that the **OPEN_MAX** file descriptors are currently open in the calling process.
ENFILE	Indicates that the maximum allowable number of files are currently open in the system.
ENXIO	Indicates that the calling process has no controlling terminal.
ERANGE	Indicates that the value of *Length* is smaller than the length of the string to be returned, including the terminating null character.

File

Item	Description
/etc/utmp	Contains a record of users logged into the system.

getmax_sl, getmax_tl, getmin_sl, and getmin_tl Subroutines

Purpose

Retrieve maximum and minimum sensitivity label (SL) and integrity label (TL) from the initialized label encodings file.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>
int getmax_sl (sl)
sl_t *sl;

int getmax_tl (tl)
tl_t *tl;

int getmin_sl(sl)
sl_t *sl;

int getmin_tl(tl)
sl_t *tl;
```

Description

The **getmax_sl** subroutine retrieves the maximum SL that is defined in the initialized label encodings file and copies the result to the *sl* parameter.

The **getmax_tl** subroutine retrieves the maximum TL that is defined in the initialized label encodings file and copies the result to the *tl* parameter.

The **getmin_sl** subroutine retrieves the minimum SL that is defined in the initialized label encodings file and copies the result to the *sl* parameter.

The **getmax_tl** subroutine retrieves the minimum TL that is defined in the initialized label encodings file and copies the result to the *tl* parameter.

Requirement: Must initialize the database before running these subroutines.

Parameters

Item	Description
sl	Specifies the sensitivity label to be copied to.
tl	Specifies the integrity label to be copied to.

Files Access

Mode	File
r	**/etc/security/enc/LabelEncodings**

Return Values

If successful, these subroutines return a value of zero. Otherwise, they return a value of -1.

Error Codes

If these subroutines fail, they return one of the following error codes:

Item	Description
ENIVAL	The parameter specifies a value that is null.
ENOTREADY	The database is not initialized.

getmaxyx Subroutine

Purpose

Returns the size of a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
getmaxyx( Window, Y, X);
WINDOW *Window;
int  Y,  X;
```

Description

The **getmaxyx** subroutine returns the size of a window. The size is returned as the number of rows and columns in the window. The values are stored in integers *Y* and *X*.

Parameters

Item	Description
Window	Identifies the window whose size to get.
Y	Contains the number of rows in the window.
X	Contains the number of columns in the window.

Example

To obtain the size of the `my_win` window, use:

```
WINDOW *my_win;

int y,x;
getmaxyx(my_win, y, x);
```

Integers *y* and *x* will contain the size of the window.

getnextprojdb Subroutine

Purpose

Retrieves the next project from the specified project database.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

getnextprojdb(void *handle, struct project *project, char *comm)
```

Description

The **getnextprojdb** subroutine retrieves the next project definitions from the project database named through the *handle* parameter. The caller must initialize the project database prior to calling this routine with the **projdballoc** routine. Upon successful completion, the project information is copied to the project structure specified by the caller. In addition, the associated project comment, if present, is copied to the buffer pointed to by the *comm* parameter. The comment buffer is allocated by the caller and must have a length of 1024 bytes.

There is an internal state (that is, the current project) associated with the project database. When the project database is initialized, the current project is the first project in the database. The **getnextprojdb** subroutine returns the current project and advances the current project assignment to the next project in the database so that successive calls read each project entry in the database. When the last project is read, the current project assignment is advanced to the end of the database. Any attempt to read beyond the end of the project database results in a failure.

Parameters

Item	Description
handle	Pointer to the **projdb** handle.
project	Pointer to project structure where the retrieved data is stored.
comm	Comment associated with the project in the database.

Security

No restriction. Any user can call this function.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Invalid arguments, if passed pointer is NULL.
ENOENT	End of the project database.
ENOENT	No projects available.

getnstr, getstr, mvgetnstr, mvgetstr, mvwgetnstr, mvwgetstr, wgetnstr, or wgetstr Subroutine

Purpose

Gets a multi-byte character string from the terminal.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int getnstr(char  *str,
int n);

int getstr(char *str);

int mvgetnstr(int y,
int x,
char *st,
int n);

int mvgetstr(int y,
int x,
char *str);

int mvwgetnstr(WINDOW  *win,
int y,
int x,
char *str,
int n);

int mvwgetstr(WINDOW *win,
int y,
int x,
char *str);

int wgetnstr(WINDOW *win,
char *str,
int n);

int wgetstr(WINDOW *win,
char *str);
```

Description

The effect of the **getstr** subroutine is as though a series of calls to the **getch** subroutine was made, until a **newline** subroutine, carriage return, or end-of-file is received. The resulting value is placed in the area pointed to by *str*. The string is then terminated with a null byte. The **getnstr**, **mvgetnstr**, **mvwgetnstr**, and **wgetnstr** subroutines read at most *n* bytes, thus preventing a possible overflow of the input buffer. The user's erase and kill characters are interpreted, as well as any special keys (such as function keys, home key, clear key, and so on).

The **mvgetstr** subroutines is identical to the **getstr** subroutine except that it is as though it is a call to the **move** subroutine and then a series of calls to the **getch** subroutine. The **mvwgetstr** subroutine is identical to the **getstr** subroutine except that it is as though it is a call to the **wmove** subroutine and then a series of calls to the **wgetch** subroutine.

The **mvgetnstr** subroutines is identical to the **getstr** subroutine except that it is as though it is a call to the **move** subroutine and then a series of calls to the **getch** subroutine. The **mvwgetnstr** subroutine is identical to the **getstr** subroutine except that it is as though it is a call to the **wmove** subroutine and then a series of calls to the **wgetch** subroutine.

The **getstr**, **wgetstr**, **mvgetstr**, and **mvwgetstr** subroutines will only return the entire multi-byte sequence associated with a character. If the array is large enough to contain at least one character, the subroutines fill the array with complete characters. If the array is not large enough to contain any complete characters, the function fails.

Parameters

Item	Description
n	Specifies the upper boundary on the number of bytes to read.
x	Holds the column coordinate of the logical cursor.
y	Holds the line or row coordinate of the logical cursor.
**str*	Identifies where to store the string.
**win*	Identifies the window to get the string from and echo it into.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To get a string, store it in the user-defined variable `my_string`, and echo it into the stdscr, enter:

```
char *my_string;
getstr(my_string);
```

2. To get a string, echo it into the user-defined window `my_window`, and store it in the user-defined variable `my_string`, enter:

```
WINDOW *my_window;
char *my_string;
wgetstr(my_window, my_string);
```

3. To get a string in the stdscr at coordinates `y=20`, `x=30`, and store it in the user-defined variable `my_string`, enter:

```
char *string;
mvgetstr(20, 30, string);
```

4. To get a string in the user-defined window `my_window` at coordinates `y=20`, `x=30`, and store it in the user-defined variable `my_string`, enter:

```
WINDOW *my_window;
char *my_string;
mvwgetstr(my_window, 20, 30, my_string);
```

getobjattr Subroutine

Purpose

Queries the object security information defined in the domain-assigned object database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int getobjattr ( Obj, Attribute, Value, Type)
char * Obj;
char * Attribute;
void *Value;
int Type;
```

Description

The **getobjattr** subroutine reads a specified attribute from the domain-assigned object database. If the database is not open, this subroutine does an implicit open for reading. For attributes of the SEC_CHAR and SEC_LIST types, the **getobjattr** subroutine returns the value to the allocated memory. The caller must free this allocated memory.

Parameters

Item	Description
Obj	Specifies the object name.
Attribute	Specifies the attribute to read. The following possible attributes are defined in the **usersec.h** file: • **S_DOMAINS** The list of domains to which the object belongs. The attribute type is **SEC_LIST**. • **S_CONFSETS** The list of domains that are excluded from accessing the object. The attribute type is **SEC_LIST** • S_TYPE The type of the object. Valid values are: – S_NETINT For Network interfaces – S_FILE For file based objects. The object name should be the absolute path – S_DEVICE For Devices. The absolute path should be specified. – S_NETPORT For port and port ranges The attribute type is SEC_CHAR. • S_SECFLAGS The security flags for the object. The valid values are FSF_DOM_ALL and FSF_DOM_ANY. The attribute type is SEC_INT.
Value	Specifies a pointer, or a pointer to a pointer according to the value specified in the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Item	Description
Type	The *Type* parameter specifies the type of the attribute. The following valid types are defined in the **usersec.h** file: SEC_INT The format of the attribute is an integer. For the subroutine, you must provide a pointer to a defined integer variable. SEC_LIST The format of the attribute is a series of concatenated strings each of which is null-terminated. The last string in the series is terminated by two successive null characters. For the subroutine, you must supply a pointer to a defined character pointer variable. The caller must free this memory.

Security

Files Accessed:

Item	Description
File	**Mode**
`/etc/security/domobjs`	`rw`

Return Values

If successful, the **getobjattr** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

Item	Description
EINVAL	The *Obj* parameter is NULL. The *Attribute* or *Type* parameter is NULL or does not contain one of the defined values. The *Obj* parameter is ALL and the *Attribute* parameter is not S_DOMAINS. The *Value* parameter does not point to a valid buffer for this type of attribute.
ENOATTR	The *Attribute* parameter is S_DOMAINS, but the *Obj* parameter is not ALL. The attribute specified in the *Attribute* parameter is valid but no value is defined for the object.
ENOENT	The object specified in the *Obj* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

getobjattrs Subroutine

Purpose

Retrieves multiple object security attributes from the domain-assigned object database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int getobjattrs ( Obj, Attributes, Count)
char * Obj;
dbattr_t *Attributes;
int Count;
```

Description

The **getobjattrs** subroutine reads one or more attributes from the domain-assigned object database. The **Attributes** array contains information about each attribute that is to be read. Each element in the **Attributes** array must be examined upon a successful call to the **getobjattrs** subroutine, to determine whether the **Attributes** array was successfully retrieved. The attributes of the SEC_CHAR or SEC_LIST type will have their values returned to the allocated memory. The caller must free this memory. The **dbattr_t** data structure contains the following fields:

The name of the target object attribute. The following valid object attributes for the **getobjattrs** subroutine are defined in the **usersec.h** file:

attr_name
Specifies the name.

attr_idx
This attribute is used internally by the **getobjattrs** subroutine.

attr_type
The type of a target attribute.

attr _flag
The result of the request is to read the target attribute. On successful completion, a value of zero is returned. Otherwise, a nonzero value is returned.

attr_un
A union that contains the returned values for the requested query.

The following table lists the different vales for *attr_name* attribute:

Name	Description	Type
S_DOMAINS	A list domains of the object.	SEC_LIST
S_CONFSETS	The list of domains defined in the conflict set of the object.	SEC_LIST
S_TYPE	The type of the object. Valid values are: S_DEVICE, S_FILE, S_NETPORT, S_NETINT	SEC_CHAR
S_SECFLAGS	The security flag associated with the object. The valid values are: FSF_DOM_ALL and FSF_DOM_ANY.	SEC_INT

The following union members correspond to the definitions of the **attr_char**, **attr_int**, **attr_long** and **attr_llong** macros in the **usersec.h** file:

au_char
Attributes of the SEC_CHAR and SEC_LIST types store a pointer to the returned value in this member when the attributes are successfully retrieved. The caller is responsible for freeing this memory.

au_int
The storage location for attributes of the SEC_INT type.

au_long
The storage location for attributes of the SEC_LONG type.

au_llong
The storage location for attributes of the SEC_LLONG type.

If ALL is specified for the *Obj* parameter, the only valid attribute that can be displayed in the **Attributes** array is the S_DOMAINS attribute. Specifying any other attribute with a domain name of ALL causes the **getobjattrs** subroutine to fail.

Parameters

Obj
Specifies the object name for the **Attributes** array to read.

Attributes
A pointer to an array of zero or more elements of the type **dbattr_t**. The list of domain-assigned object attributes is defined in the **usersec.h** header file.

Count
The number of array elements in the **Attributes** array.

Security

Files Accessed:

/etc/security/domains
mode: r

Return Values

If the object specified by the *Obj* parameter exists in the domain-assigned object database, the **getobjattrs** subroutine returns the value of zero. On successful completion, the **attr_flag** attribute of each element in the **Attributes** array must be examined to determine whether it was successfully retrieved. If the specified object does not exist, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **getobjattrs** subroutine returns -1, one of the following **errno** values is set:

EINVAL

The *Obj* parameter is NULL.

The *Count* parameter is less than zero.

The **Attributes** array is NULL and the *Count* parameter is greater than zero.

The *Obj* parameter is ALL but the **Attributes** entry contains an attribute other than S_DOMAINS.

ENOENT
The object specified in the *Obj* parameter does not exist.

ENOMEM
Memory cannot be allocated.

EPERM
The operation is not permitted.

EACCES
Access permission is denied for the data request.

If the **getobjattrs** subroutine fails to query an attribute, one of the following errors is returned to the **attr_flag** field of the corresponding **Attributes** element:

EACCES
The invoker does not have access to the attribute specified in the **attr_name** field.

EINVAL

The **attr_name** field in the **Attributes** entry is not a recognized object attribute.

The **attr_type** field in the **Attributes** entry contains a type that is not valid.

The **attr_un** field in the **Attributes** entry does not point to a valid buffer.

ENOATTR

The **attr_name** field in the **Attributes** entry specifies a valid attribute, but no value is defined for this object.

getopt Subroutine

Purpose

Returns the next flag letter specified on the command line.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int getopt (ArgumentC,  ArgumentV,  OptionString)
int ArgumentC;
char *const ArgumentV [ ];
const char *OptionString;

extern int  optind;
extern int  optopt;
extern int  opterr;
extern char * optarg;
```

Description

The *optind* parameter indexes the next element of the *ArgumentV* parameter to be processed. It is initialized to 1 and the **getopt** subroutine updates it after calling each element of the *ArgumentV* parameter.

The **getopt** subroutine returns the next flag letter in the *ArgumentV* parameter list that matches a letter in the *OptionString* parameter. If the flag takes an argument, the **getopt** subroutine sets the *optarg* parameter to point to the argument as follows:

- If the flag was the last letter in the string pointed to by an element of the *ArgumentV* parameter, the *optarg* parameter contains the next element of the *ArgumentV* parameter and the *optind* parameter is incremented by 2. If the resulting value of the *optind* parameter is not less than the *ArgumentC* parameter, this indicates a missing flag argument, and the **getopt** subroutine returns an error message.

- Otherwise, the *optarg* parameter points to the string following the flag letter in that element of the *ArgumentV* parameter and the *optind* parameter is incremented by 1.

Note: The user who wants to scan the same *ArgumentV* parameter again or scan multiple *ArgumentV* sets in the same program, need to reinitialize the **getopt()** subroutine by setting the *optind* parameter to 0.

Parameters

Item	Description
ArgumentC	Specifies the number of parameters passed to the routine.
ArgumentV	Specifies the list of parameters passed to the routine.
OptionString	Specifies a string of recognized flag letters. If a letter is followed by a : (colon), the flag is expected to take a parameter that may or may not be separated from it by white space.
optind	Specifies the next element of the *ArgumentV* array to be processed.
optopt	Specifies any erroneous character in the *OptionString* parameter.
opterr	Indicates that an error has occurred when set to a value other than 0.
optarg	Points to the next option flag argument.

Return Values

The **getopt** subroutine returns the next flag letter specified on the command line. A value of -1 is returned when all command line flags have been parsed. When the value of the *ArgumentV* **[***optind***]** parameter is null, ***ArgumentV* [*optind*]** is not the **-** (minus) character, or *ArgumentV* **[***optind***]** points to the "**-**" (minus) string, the **getopt** subroutine returns a value of -1 without changing the value. If *ArgumentV* **[***optind***]** points to the "**- -**" (double minus) string, the **getopt** subroutine returns a value of -1 after incrementing the value of the *optind* parameter.

Error Codes

If the **getopt** subroutine encounters an option character that is not specified by the *OptionString* parameter, a **?** (question mark) character is returned. If it detects a missing option argument and the first character of *OptionString* is a **:** (colon), then a **:** (colon) character is returned. If this subroutine detects a missing option argument and the first character of *OptionString* is not a colon, it returns a **?** (question mark). In either case, the **getopt** subroutine sets the *optopt* parameter to the option character that caused the error. If the application has not set the *opterr* parameter to 0 and the first character of *OptionString* is not a **:** (colon), the **getopt** subroutine also prints a diagnostic message to standard error.

Examples

The following code fragment processes the flags for a command that can take the mutually exclusive flags **a** and **b**, and the flags **f** and **o**, both of which require parameters.

```
#include <unistd.h>    /*Needed for access subroutine constants*/
main (argc, argv)
int argc;
char **argv;
{
   int c;
   extern int optind;
   extern char *optarg;
   .
   .
   .
   while ((c = getopt(argc, argv, "abf:o:")) != EOF)

   {
      switch (c)
```

```
      {
         case 'a':
            if (bflg)
               errflg++;
            else
               aflg++;
            break;

         case 'b':
            if (aflg)
               errflg++;
            else
               bflg++;
            break;

         case 'f':
            ifile = optarg;
            break;

         case 'o':
            ofile = optarg;
            break;

         case '?':
            errflg++;
      } /* case */

      if (errflg)
      {
         fprintf(stderr, "usage: . . . ");
         exit(2);
      }
   } /* while */

   for ( ; optind < argc; optind++)
   {
      if (access(argv[optind], R_OK))
      {
         .
         .
         .
      }
   } /* for */
}   /* main */
```

getosuuid Subroutine

Purpose

Retrieves the operating system Universal Unique Identifier (UUID).

Library

Standard C Library (**libc.a**)

Syntax

```
#include <uuid.h>
int getosuuid (uuid,uuid_type)
uuid_t * uuid;
int uuid_type;
```

Description

Retrieves the operating system UUID saved in the AIX kernel. If in a WPAR, the WPAR UUID is returned instead.

Note:

The UUID of the AIX operating system can be retrieved using the **lsattr** command:

```
lsattr -l sys0 -a os_uuid -E
```

Parameters

Item	Description
uuid	Points to the location used to return the operating system UUID.
uuid_type	Specifies the type of UUID to retrieve. Must be **GETOSUUID_AIX**.

Return Values

Upon successful completion the **getosuuid** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EINVAL	Indicates the value of the *uuid_type* parameter is invalid.
EFAULT	Invalid address in parameter *uuid*.

getpagesize Subroutine

Purpose

Gets the system page size.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int getpagesize( )
```

Description

The **getpagesize** subroutine returns the number of bytes in a page. Page granularity is the granularity for many of the memory management calls.

The page size is determined by the system and may not be the same as the underlying hardware page size.

getpaginfo Subroutine

Purpose

Retrieves a Process Authentication Group (PAG) flags for a given PAG type.

Library

Security Library (`libc.a`)

Syntax

```
#include <pag.h>

int getpaginfo ( name, infop, infosz )
char * name;
struct paginfo * infop;
int infosz;
```

Description

The `getpaginfo` subroutine retrieves the PAG flags for a given PAG name. For this function to succeed, the PAG name must be registered with the operating system before this subroutine is called. The *infop* parameter must be a valid, referenced PAG info structure of the size specified by *infosz*.

Parameters

Item	Description
name	A 1-character to 4-character, NULL-terminated name for the PAG type. Typical values include `afs`, `dfs`, `pki`, and `krb5`.
infop	Points to a `paginfo struct` where the operating system returns the PAG flags.
infosz	Indicates the size of the PAG info structure.

Return Values

A value of `0` is returned upon successful completion. If the `getpaginfo` subroutine fails a value of `-1` is returned and the `errno` global variable is set to indicate the error.

Error Codes

The `getpaginfo` subroutine fails if the following condition is true:

Item	Description
`EINVAL`	The named PAG type does not exist as part of the table.

Other errors might be set by subroutines invoked by the `getpaginfo` subroutine.

getpagvalue or getpagvalue64 Subroutine

Purpose

Returns the Process Authentication Group (PAG) value for a given PAG type.

Library

Security Library (`libc.a`)

Syntax

```
#include <pag.h>

int getpagvalue ( name )
char * name;

uint64_t getpagvalue64( name );
char * name;
```

Description

The `getpagvalue` and `getpagvalue64` subroutines retrieve the PAG value for a given PAG name. For these functions to succeed, the PAG name must be registered with the operating system before these subroutines are called.

Parameters

Item	Description
name	A 1-character to 4-character, NULL-terminated name for the PAG type. Typical values include `afs`, `dfs`, `pki`, and `krb5`.

Return Values

The `getpagvalue` and `getpagvalue64` subroutines return a PAG value upon successful completion. Upon a failure, a value of `-1` is returned and the `errno` global variable is set to indicate the error.

Error Codes

The `getpagvalue` and `getpagvalue64` subroutines fail if the following condition is true:

Item	Description
`EINVAL`	The named PAG type does not exist as part of the table.

Other errors might be set by subroutines invoked by the `getpagvalue` and `getpagvalue64` subroutines.

getpass Subroutine

Purpose

Reads a password.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

char *getpass ( Prompt)
char *Prompt;
```

Description

Attention: The characters are returned in a static data area. Subsequent calls to this subroutine overwrite the static data area.

The **getpass** subroutine does the following:

- Opens the controlling terminal of the current process.
- Writes the characters specified by the *Prompt* parameter to that device.
- Reads from that device the number of characters up to the value of the **PASS_MAX** constant until a new-line or end-of-file (EOF) character is detected.
- Restores the terminal state and closes the controlling terminal.

During the read operation, character echoing is disabled.

The **getpass** subroutine is not safe in a multithreaded environment. To use the **getpass** subroutine in a threaded application, the application must keep the integrity of each thread.

Parameters

Item	Description
Prompt	Specifies a prompt to display on the terminal.

Return Values

If this subroutine is successful, it returns a pointer to the string. If an error occurs, the subroutine returns a null pointer and sets the **errno** global variable to indicate the error.

Error Codes

If the **getpass** subroutine is unsuccessful, it returns one or more of the following error codes:

Item	Description
EINTR	Indicates that an interrupt occurred while the **getpass** subroutine was reading the terminal device. If a **SIGINT** or **SIGQUIT** signal is received, the **getpass** subroutine terminates input and sends the signal to the calling process.
ENXIO	Indicates that the process does not have a controlling terminal.

Note: Any subroutines called by the **getpass** subroutine may set other error codes.

getpcred Subroutine

Purpose

Reads the current process credentials.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

char **getpcred ( Which)
int Which;
```

Description

The **getpcred** subroutine reads the specified process security credentials and returns a pointer to a NULL terminated array of pointers in allocated memory. Each pointer in the array points to a string containing an attribute/value pair in allocated memory. It's the responsibility of the caller to free each individual string as well as the array of pointers.

Parameters

Item	Description
Which	Specifies which credentials are read. This parameter is a bit mask and can contain one or more of the following values, as defined in the **usersec.h** file: **CRED_RUID** Real user name **CRED_LUID** Login user name **CRED_RGID** Real group name **CRED_GROUPS** Supplementary group ID **CRED_AUDIT** Audit class of the current process **Note:** A process must have root user authority to retrieve this credential. Otherwise, the **getpcred** subroutine returns a null pointer and the **errno** global variable is set to **EPERM**. **CRED_RLIMITS** BSD resource limits **Note:** Use the **getrlimit** (“getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine” on page 526) subroutine to control resource consumption. **CRED_UMASK** The umask. If the *Which* parameter is null, all credentials are returned.

Return Values

When successful, the **getpcred** subroutine returns a pointer to a NULL terminated array of string pointers containing the requested values. If the **getpcred** subroutine is unsuccessful, a NULL pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getpcred** subroutine fails if either of the following are true:

Item	Description
EINVAL	The *Which* parameter contains invalid credentials requests.
EPERM	The process does not have the proper authority to retrieve the requested credentials.

Other errors can also be set by any subroutines invoked by the **getpcred** subroutine.

getpeereid Subroutine

Note: The **getpeerid** technology used to support this function in AIX was originally published by D. J. Bernstein, Associate Professor, Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago. In addition, the specific **getpeerid** syntax reflected originated with William Erik Baxter. All the aforementioned are used by AIX with permission.

Purpose

Gets the effective user ID and effective group ID of a peer on a connected UNIX domain socket.

Syntax

```
#include <sys/types.h>
    int getpeereid (int socket, uid_t *euid, gid_t *egid)
```

Description

The **getpeereid** subroutine returns the effective user and group IDs of the peer connected to a stream socket in the UNIX domain. The effective user and group IDs are saved in the socket, to be returned, when the peer calls **connect** or **listen**.

Parameters

Item	Description
socket	Specifies the descriptor number of a connected socket.
euid	The effective user ID of the peer socket.
egid	The effective group ID of the peer socket.

Return Values

When the **getpeereid** subroutine successfully completes, a value of 0 is returned and the *euid* and *egid* parameters hold the effective user ID and group ID, respectively.

If the **getpeereid** subroutine is unsuccessful, the system handler returns a value of -1 to the calling program and sets the **errno** global variable to an error code that indicates the specific error.

Error Codes

The **getpeereid** subroutine is unsuccessful if any of the following errors occurs:

Item	Description
EBADF	The *socket* parameter is not valid.
ENOTSOCK	The *socket* parameter refers to a file, not a socket.
ENOTCONN	The socket is not connected.
ENOBUFS	Insufficient resources were available in the system to complete the call.
EFAULT	The *address* parameter is not in a writable part of the user address space.

getpenv Subroutine

Purpose

Reads the current process environment.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

char **getpenv ( Which)
int Which;
```

Description

The **getpenv** subroutine reads the specified environment variables and returns them in a character buffer.

Parameters

Item	Description
Which	Specifies which environment variables are to be returned. This parameter is a bit mask and may contain one or more of the following values, as defined in the **usersec.h** file: **PENV_USR** The normal user-state environment. Typically, the shell variables are contained here. **PENV_SYS** The system-state environment. This data is located in system space and protected from unauthorized access. All variables are returned by setting the *Which* parameter to logically OR the **PENV_USER** and **PENV_SYSTEM** values. The variables are returned in a null-terminated array of character pointers in the form `var=val`. The user-state environment variables are prefaced by the string **USRENVIRON:**, and the system-state variables are prefaced with **SYSENVIRON:**. If a user-state environment is requested, the current directory is always returned in the **PWD** variable. If this variable is not present in the existing environment, the **getpenv** subroutine adds it to the returned string.

Return Values

Upon successful return, the **getpenv** subroutine returns the environment values. If the **getpenv** subroutine fails, a null value is returned and the **errno** global variable is set to indicate the error.

Note: This subroutine can partially succeed, returning only the values that the process permits it to read.

Error Codes

The **getpenv** subroutine fails if one or more of the following are true:

Item	Description
EINVAL	The *Which* parameter contains values other than **PENV_USR** or **PENV_SYS**.

Other errors can also be set by subroutines invoked by the **getpenv** subroutine.

getpfileattr Subroutine

Purpose

Accesses the privileged file security information in the privileged file database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getpfileattr (File, Attribute, Value, Type)
    char *File;
    char *Attribute;
    void *Value;
    int Type;
```

Description

The **getpfileattr** subroutine reads a specified attribute from the privileged file database. If the database is not open, this subroutine does an implicit open for reading.

Parameters

Item	Description
File	Specifies the file name. The value must be the full path to the file on the system. This parameter must be specified unless the value of the *Type* parameter is **SEC_COMMIT**.
Attribute	Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file: **S_READAUTHS** Authorizations required to read the file using the **pvi** command. A total of eight authorizations can be defined. The attribute type is **SEC_LIST**. **S_WRITEAUTHS** Authorizations required to write to the file using the **pvi** command. A total of eight authorizations can be defined. The attribute type is **SEC_LIST**.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.
Type	Specifies the type of attribute expected. The **usersec.h** file defines and includes the following valid types: **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters. For the **getpfileattr** subroutine, you must supply a pointer to a defined character pointer variable. It is the caller's responsibility to free this memory. **SEC_DELETE** If the *Attribute* parameter is specified, the corresponding attribute is deleted from the privileged file database. If no *Attribute* parameter is specified, the entire privileged file definition is deleted from the privileged file database.

Security

Files Accessed:

File	Mode
/etc/security/privfiles	rw

Return Values

If successful, the **getpfileattr** subroutine returns a value of zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **getpfileattr** subroutine fails, one of the following **errno** values can be set:

Item	Description
EINVAL	The *File* parameter is **NULL** or **default**.
EINVAL	The *Attribute* or *Type* parameter is **NULL** or does not contain one of the defined values.
EINVAL	The *Attribute* parameter is **S_PRIVFILES**, but the *File* parameter is not **ALL**.
EINVAL	The *Value* parameter does not point to a valid buffer for this type of attribute.
ENOENT	The file specified in the *File* parameter does not exist.
ENOATTR	The attribute specified in the *Attribute* parameter is valid, but no value is defined for the file.
EPERM	Operation is not permitted.

getpfileattrs Subroutine

Purpose

Retrieves multiple file attributes from the privileged file database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getpfileattrs(File, Attributes, Count)
    char *File;
    dbattr_t *Attributes;
    int Count;
```

Description

The **getpfileattrs** subroutine reads one or more attributes from the privileged file database (**/etc/security/privfiles**). The file specified with the *File* parameter must include the full path to the file and exist in the privileged file database. If the database is not open, this subroutine does an implicit open for reading.

The Attributes array contains information about each attribute that is to be read. Each element in the Attributes array must be examined upon a successful call to the **getpfileattrs** subroutine to determine

whether the Attributes array was successfully retrieved. The **dbattr_t data** structure contains the following fields:

Item	Description
attr_name	The name of the desired file attribute.
attr_idx	This attribute is used internally by the **getpfileattrs** subroutine.
attr_type	The type of the target attribute.
attr _flag	The result of the request to read the target attribute. A value of zero is returned on success; a nonzero value is returned otherwise.
attr_un	A union containing the returned values for the requested query.

Valid privileged file attributes for the **getpfileattrs** subroutine defined in the **usersec.h** file are:

Name	Description	Type
S_PRIVFILES	Retrieves all the files in the privileged file database. It is valid only when the *File* parameter is **ALL**.	**SEC_LIST**
S_READAUTHS	Read authorization. It is a null separated list of authorization names. A total of eight authorizations can be specified. A user with any one of the authorizations is allowed to read the file using the privileged editor **/usr/bin/pvi**.	**Steeliest**
S_WRITEAUTHS	Write authorization. It is a null separated list of authorization names. A total of eight authorizations can be specified. A user with any one of the authorizations is allowed to write the file using the privileged editor **/usr/bin/pvi**.	**SEC_LIST**

The union members that follow correspond to the definitions of the **attr_char, attr_int, attr_long** and **attr_llong** macros in the **usersec.h** file respectively.

Item	Description
au_char	Attributes of the **SEC_CHAR** and **SEC_LIST** types store a pointer to the returned value in this member when the attributes are successfully retrieved. The caller is responsible for freeing this memory.
au_int	Storage location for attributes of the **SEC_INT** type.
au_long	Storage location for attributes of the **SEC_LONG** type.
au_llong	Storage location for attributes of the **SEC_LLONG** type.

If **ALL** is specified for the *File* parameter, the only valid attribute that can appear in the Attribute array is the **S_PRIVFILES** attribute. Specifying any other attribute with a file name of **ALL** causes the **getpfileattrs** subroutine to fail.

Parameters

Item	Description
File	Specifies the file name for which the attributes are to be read.
Attributes	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of file attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the Attributes array.

Security

Files Accessed:

File	Mode
/etc/security/privfiles	r

Return Values

If the file specified by the *File* parameter exists in the privileged file database, the **getpfileattrs** subroutine returns zero. On success, the **attr_flag** attribute of each element in the Attributes array must be examined to determine whether it was successfully retrieved. If the specified file does not exist, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **getpfileattrs** subroutine returns -1, one of the following **errno** values can be set:

Item	Description
EINVAL	The *File* parameter is **NULL** or **default**.
EINVAL	The *File* parameter is **ALL** but the Attributes entry contains an attribute other than **S_PRIVFILES**.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *File* parameter is **NULL** and the *Count* parameter is greater than zero.
ENOENT	The file specified in the *File* parameter does not exist in the database.
EPERM	Operation is not permitted.

If the **getpfileattrs** subroutine fails to query an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding Attributes element:

Item	Description
EACCES	The invoker does not have access to the attribute specified in the **attr_name** field.
EINVAL	The **attr_name** field in the Attributes entry is not a recognized file attribute.
EINVAL	The **attr_type** field in the Attributes entry contains an invalid type.
EINVAL	The **attr_un** field in the Attributes entry does not point to a valid buffer.
ENOATTR	The **attr_name** field in the Attributes entry specifies a valid attribute, but no value is defined for this file.
ENOMEM	Memory cannot be allocated to store the return value.

getpgid Subroutine

Purpose

Returns the process group ID of the calling process.

Library

Standard C Library **(libc.a)**

Syntax

#include <unistd.h>

```
pid_t getpgid (Pid)
(pid_ Pid)
```

Description

The **getpgid** subroutine returns the process group ID of the process whose process ID is equal to that specified by the *Pid* parameter. If the value of the *Pid* parameter is equal to **(pid_t)0**, the **getpgid** subroutine returns the process group ID of the calling process.

Parameter

Item	Description
Pid	The process ID of the process to return the process group ID for.

Return Values

Item	Description
id	The process group ID of the requested process
-1	Not successful and **errno** set to one of the following.

Error Code

Item	Description
ESRCH	There is no process with a process ID equal to *Pid*.

Item	Description
EPERM	The process whose process ID is equal to *Pid* is not in the same session as the calling process.
EINVAL	The value of the *Pid* argument is invalid.

getpid, getpgrp, or getppid Subroutine

Purpose

Returns the process ID, process group ID, and parent process ID.

Syntax

```
#include <unistd.h>
```

```
pid_t getpid (void)
```

```
pid_t getpgrp (void)
```

```
pid_t getppid (void)
```

Description

The **getpid** subroutine returns the process ID of the calling process.

The **getpgrp** subroutine returns the process group ID of the calling process.

The **getppid** subroutine returns the process ID of the calling process' parent process.

getportattr or putportattr Subroutine

Purpose

Accesses the port information in the port database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int getportattr (Port, Attribute, Value, Type)
char * Port;
char * Attribute;
void * Value;
int  Type;
```

```
int putportattr (Port, Attribute, Value, Type)
char *Port;
char *Attribute;
void *Value;
int Type;
```

Description

The **getportattr** or **putportattr** subroutine accesses port information. The **getportattr** subroutine reads a specified attribute from the port database. If the database is not already open, the **getportattr** subroutine implicitly opens the database for reading. The **putportattr** subroutine writes a specified attribute into the port database. If the database is not already open, the **putportattr** subroutine implicitly opens the database for reading and writing. The data changed by the **putportattr** subroutine must be explicitly committed by calling the **putportattr** subroutine with a *Type* parameter equal to the **SEC_COMMIT** value. Until all the data is committed, only these subroutines within the process return the written data.

Values returned by these subroutines are in dynamically allocated buffers. You do not need to move the values prior to the next call.

Use the **setuserdb** or **enduserdb** subroutine to open and close the port database.

Parameters

Item	Description
Port	Specifies the name of the port for which an attribute is read.
Attribute	Specifies the name of the attribute read. This attribute can be one of the following values defined in the **usersec.h** file: **S_HERALD** Defines the initial message printed when the **getty** or **login** command prompts for a login name. This value is of the type **SEC_CHAR**. **S_SAKENABLED** Indicates whether or not trusted path processing is allowed on this port. This value is of the type **SEC_BOOL**. **S_SYNONYM** Defines the set of ports that are **synonym** attributes for the given port. This value is of the type **SEC_LIST**. **S_LOGTIMES** Defines when the user can access the port. This value is of the type **SEC_LIST**. **S_LOGDISABLE** Defines the number of unsuccessful login attempts that result in the system locking the port. This value is of the type **SEC_INT**. **S_LOGINTERVAL** Defines the time interval in seconds within which **S_LOGDISABLE** number of unsuccessful login attempts must occur before the system locks the port. This value is of the type **SEC_INT**. **S_LOGREENABLE** Defines the time interval in minutes after which a system-locked port is unlocked. This value is of the type **SEC_INT**. **S_LOGDELAY** Defines the delay factor in seconds between unsuccessful login attempts. This value is of the type **SEC_INT**. **S_LOCKTIME** Defines the time in seconds since the epoch (zero time, January 1, 1970) that the port was locked. This value is of the type **SEC_INT**. **S_ULOGTIMES** Lists the times in seconds since the epoch (midnight, January 1, 1970) when unsuccessful login attempts occurred. This value is of the type **SEC_LIST**. **S_USERNAMEECHO** Indicates whether user name input echo and user name masking is enabled for the port. This value is of the type **SEC_BOOL**. **S_PWDPROMPT** Defines the password prompt message printed when requesting password input. This value is of the type **SEC_CHAR**.
Value	Specifies the address of a buffer in which the attribute is stored with **putportattr** or is to be read **getportattr**.

Item	Description
Type	Specifies the type of attribute expected. The following types are valid and defined in the **usersec.h** file: **SEC_INT** Indicates the format of the attribute is an integer. The buffer returned by the **getportattr** subroutine and the buffer supplied by the **putportattr** subroutine are defined to contain an integer. **SEC_CHAR** Indicates the format of the attribute is a null-terminated character string. **SEC_LIST** Indicates the format of the attribute is a list of null-terminated character strings. The list itself is null terminated. **SEC_BOOL** An integer with a value of either 0 or 1, or a pointer to a character pointing to one of the following strings: • True • Yes • Always • False • No • Never **SEC_COMMIT** Indicates that changes to the specified port are committed to permanent storage if specified alone for the **putportattr** subroutine. The *Attribute* and *Value* parameters are ignored. If no port is specified, changes to all modified ports are committed. **SEC_DELETE** Deletes the corresponding attribute from the database. **SEC_NEW** Updates all of the port database files with the new port name when using the **putportattr** subroutine.

Security

Access Control: The calling process must have access to the port information in the port database.

File Accessed:

Item	Description
rw	/etc/security/login.cfg
rw	/etc/security/portlog

Return Values

The **getportattr** and **putportattr** subroutines return a value of 0 if completed successfully. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

These subroutines are unsuccessful if the following values are true:

Item	Description
EACCES	Indicates that access permission is denied for the data requested.
ENOENT	Indicates that the *Port* parameter does not exist or the attribute is not defined for the specified port.
ENOATTR	Indicates that the specified port attribute does not exist for the specified port.
EINVAL	Indicates that the *Attribute* parameter does not contain one of the defined attributes or is a null value.
EINVAL	Indicates that the *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.

Item	Description
EPERM	Operation is not permitted.

getppriv Subroutine

Purpose

Gets a privilege set associated with a process.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/priv.h>
int getppriv(pid, which, privset, privsize)
pid_t pid;
int which;
privg_t *privset;
int privsize;
```

Description

The **getppriv** subroutine returns the privilege set for the process specified by the *pid* parameter. If the value of the *pid* is negative, the calling process's privilege set is retrieved. The value of the *which* parameter is one of the PRIV_EFFECTIVE, PRIV_MAXIMUM, PRIV_INHERITED, PRIV_LIMITING or PRIV_USED values. The corresponding privilege set is copied to the *privset* parameter in the size specified by the *privsize* parameter. The PV_PROC_PRIV privilege is required in the effective set when a process wants to obtain privilege set from another process.

Parameters

Item	Description
Pid	Indicates the process that the privilege set is requested for.
Which	Specifies the privilege set to get.
Privset	Stores the privilege set.
Privsize	Specifies the size of the privilege set.

Return Values

The **getppriv** subroutine returns one of the following values:

Item	Description
0	The subroutine completes successfully.
-1	An error has occurred. An **errno** global variable is set to indicate the error.

Error Codes

The **getppriv** subroutine fails if any of the following values is true:

Item	Description
EFAULT	The *privset* parameter is pointing to an address that is not legal.
EINVAL	The value of the *privset* parameter is NULL, or the value of the *privsize* parameter is not valid.
EPERM	The process does not have the privilege (**PV_PROC_PRIV** or **MAC** read) to obtain another process' privilege set.
ESRCH	No process has a process ID that is equal to the value of the *Pid* parameter.

getpri Subroutine

Purpose

Returns the scheduling priority of a process.

Library

Standard C Library (**libc.a**)

Syntax

```
int getpri ( ProcessID)
pid_t ProcessID;
```

Description

The **getpri** subroutine returns the scheduling priority of a process.

Parameters

Item	Description
ProcessID	Specifies the process ID. If this value is 0, the current process scheduling priority is returned.

Return Values

Upon successful completion, the **getpri** subroutine returns the scheduling priority of a thread in the process. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **getpri** subroutine is unsuccessful if one of the following is true:

Item	Description
EPERM	A process was located, but its effective and real user ID did not match that of the process running the **getpri** subroutine, and the calling process did not have root user authority.
ESRCH	No process can be found corresponding to that specified by the *ProcessID* parameter.

getprivid Subroutine

Purpose

Converts a privilege name into a numeric value.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

int getprivid(char *privname)
```

Description

The **getprivid** subroutine converts a given privilege name specified by the *privname* parameter into a numeric value of the privilege index that is defined in the <**sys/priv.h**> header file.

Parameters

Item	Description
privname	Specifies the privilege name that is in string format.

Return Values

The **getprivid** subroutine returns one of the following values:

Item	Description
privilege index	The subroutine successfully completes.
-1	The subroutine cannot find the privilege name specified by the *privname* parameter.

Errors

No **errno** value is set.

getprivname Subroutine

Purpose

Converts a privilege bit into a readable string.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

char *getprivname(int priv)
```

Description

The **getprivname** subroutine converts a given privilege bit specified by the *priv* parameter into a readable string.

Parameters

Item	Description
priv	Specifies the privilege to be converted.

Return Values

The **getprivname** subroutine returns one of the following values:

Item	Description
character string	The privilege is valid.
NULL	The privilege is not valid.

Errors

No **errno** value is set.

getpriority, setpriority, or nice Subroutine

Purpose

Gets or sets the nice value.

Libraries

getpriority, **setpriority**: Standard C Library (**libc.a**)

nice: Standard C Library (**libc.a**)

Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <sys/resource.h>
```

```
int getpriority( Which,  Who)
int Which;
int Who;

int setpriority(Which, Who,  Priority)
int Which;
int Who;
int Priority;
```

```
#include <unistd.h>
```

```
int nice( Increment)
int Increment;
```

Description

The nice value of the process, process group, or user, as indicated by the *Which* and *Who* parameters is obtained with the **getpriority** subroutine and set with the **setpriority** subroutine.

The **getpriority** subroutine returns the highest priority nice value (lowest numerical value) pertaining to any of the specified processes. The **setpriority** subroutine sets the nice values of all of the specified processes to the specified value. If the specified value is less than -20, a value of -20 is used; if it is greater than 20, a value of 20 is used. Only processes that have root user authority can lower nice values.

The **nice** subroutine increments the nice value by the value of the *Increment* parameter.

Note: Nice values are only used for the scheduling policy **SCHED_OTHER**, where they are combined with a calculation of recent cpu usage to determine the priority value.

To provide upward compatibility with older programs, the **nice** interface, originally found in AT&T System V, is supported.

Note: Process priorities in AT&T System V are defined in the range of 0 to 39, rather than -20 to 20 as in BSD, and the **nice** library routine is supported by both. Accordingly, two versions of the **nice** are supported by AIX Version 3. The default version behaves like the AT&T System V version, with the *Increment* parameter treated as the modifier of a value in the range of 0 to 39 (0 corresponds to -20, 39 corresponds to 9, and priority 20 is not reachable with this interface).

If the behavior of the BSD version is desired, compile with the Berkeley Compatibility Library (**libbsd.a**). The *Increment* parameter is treated as the modifier of a value in the range -20 to 20.

Parameters

Item	Description
Which	Specifies one of **PRIO_PROCESS**, **PRIO_PGRP**, or **PRIO_USER**.
Who	Interpreted relative to the *Which* parameter (a process identifier, process group identifier, and a user ID, respectively). A zero value for the *Who* parameter denotes the current process, process group, or user.
Priority	Specifies a value in the range -20 to 20. Negative nice values cause more favorable scheduling.
Increment	Specifies a value that is added to the current process nice value. Negative values can be specified, although values exceeding either the high or low limit are truncated.

Return Values

On successful completion, the **getpriority** subroutine returns an integer in the range -20 to 20. A return value of -1 can also indicate an error, and in this case the **errno** global variable is set.

On successful completion, the **setpriority** subroutine returns 0. Otherwise, -1 is returned and the global variable **errno** is set to indicate the error.

On successful completion, the **nice** subroutine returns the new nice value minus {NZERO}. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Note: A value of -1 can also be returned. In that case, the calling process should also check the **errno** global variable.

Error Codes

The **getpriority** and **setpriority** subroutines are unsuccessful if one of the following is true:

Item	Description
ESRCH	No process was located using the *Which* and *Who* parameter values specified.
EINVAL	The *Which* parameter was not recognized.

In addition to the errors indicated above, the **setpriority** subroutine is unsuccessful if one of the following is true:

Item	Description
EPERM	A process was located, but neither the effective nor real user ID of the caller of the process executing the **setpriority** subroutine has root user authority.
EACCES	The call to **setpriority** would have changed the priority of a process to a value lower than its current value, and the effective user ID of the process executing the call did not have root user authority.

The **nice** subroutine is unsuccessful if the following is true:

Item	Description
EPERM	The *Increment* parameter is negative and the calling process does not have appropriate privileges.

getproclist, getlparlist, or getarmlist Subroutine

Purpose

Retrieve the transaction records from the advanced accounting data file.

Library

The `libaacct.a` library.

Syntax

```
#include <sys/aacct.h>
int getproclist(filename, begin_time, end_time, p_list);
int getlparlist(filename, begin_time, end_time, l_list);
int getarmlist(filename, begin_time, end_time, t_list);
char *filename;
long long begin_time;
long long end_time;
struct aacct_tran **p_list, **l_list, **t_list
```

Description

The `getproclist`, `getlparlist`, and `getarmlist` subroutines parse the specified advanced accounting data file and retrieve the process, LPAR, and ARM transaction records, respectively. The retrieved transaction records are returned in the form of a linked list of type `struct aacct_tran_rec`.

These APIs can be called multiple times with different accounting data file names in order to generate a consolidated list of transaction records from multiple data files. They append the new file data to the end of the linked list pointed to by the *p_list*, *l_list*, and *t_list* arguments. They also internally sort the transaction records based on the time of transaction, which gives users a time-sorted list of transaction records from these routines.

The `getproclist`, `getlparlist`, and `getarmlist` subroutines can also be used to retrieve the intended transaction records for a particular interval of time by passing the begin and end times of the interval as arguments to these routines. If these interval arguments are specified as `-1`, transaction records for all the intervals are retrieved.

Parameters

Item	Description
begin_time	Specifies the start timestamp for collecting records in a particular intervals. The input is in seconds since EPOCH. Specifying `-1` retrieves all the records.
end_time	Specifies the end timestamp for collecting records in a particular intervals. The input is in seconds since EPOCH. Specifying `-1` retrieves all the records.
filename	Name of the advanced accounting data file.
l_list	Pointers to the linked list of `aacct_tran_rec` structures, which hold the retrieved LPAR records.
p_list	Pointers to the linked list of `aacct_tran_rec` structures, which hold the retrieved process records.
t_list	Pointers to the linked list of `aacct_tran_rec` structures, which hold the retrieved ARM records.

Security

No restrictions. Any user can call this function.

Return Values

Item	Description
0	The call to the subroutine was successful.
-1	The call to the subroutine failed.

Error Codes

Item	Description
`EINVAL`	The passed pointer is NULL.
`ENOENT`	Specified data file does not exist.
`EPERM`	Permission denied. Unable to read the data file.
`ENOMEM`	Insufficient memory.

getprocs Subroutine

Purpose

Gets process table entries.

Library

Standard C library (**libc.a**)

Syntax

```
#include <procinfo.h>
#include <sys/types.h>
```

```
int
getprocs ( ProcessBuffer,  ProcessSize,  FileBuffer,  FileSize,  IndexPointer,  Count)
struct procsinfo *ProcessBuffer;
or struct procsinfo64 *ProcessBuffer;
int ProcessSize;
struct fdsinfo *FileBuffer;
int FileSize;
pid_t *IndexPointer;
int Count;
```

```
int
getprocs64 ( ProcessBuffer,  ProcessSize,  FileBuffer,  FileSize,  IndexPointer,  Count)
struct procentry64 *ProcessBuffer;
int ProcessSize;
struct fdsinfo64 *FileBuffer;
int FileSize;
pid_t *IndexPointer;
int Count;
```

Description

The **getprocs** subroutine returns information about processes, including process table information defined by the **procsinfo** structure, and information about the per-process file descriptors defined by the **fdsinfo** structure.

The **getprocs** subroutine retrieves up to *Count* process table entries, starting with the process table entry corresponding to the process identifier indicated by *IndexPointer*, and places them in the array of **procsinfo** structures indicated by the *ProcessBuffer* parameter. File descriptor information corresponding to the retrieved processes are stored in the array of **fdsinfo** structures indicated by the *FileBuffer* parameter.

On return, the process identifier referenced by *IndexPointer* is updated to indicate the next process table entry to be retrieved. The **getprocs** subroutine returns the number of process table entries retrieved.

The **getprocs** subroutine is normally called repeatedly in a loop, starting with a process identifier of zero, and looping until the return value is less than *Count*, indicating that there are no more entries to retrieve.

Note: The process table may change while the **getprocs** subroutine is accessing it. Returned entries will always be consistent, but since processes can be created or destroyed while the **getprocs** subroutine is running, there is no guarantee that retrieved entries will still exist, or that all existing processes have been retrieved.

When used in 32-bit mode, limits larger than can be represented in 32 bits are truncated to RLIM_INFINITY. Large **rusage** and other values are truncated to INT_MAX. Alternatively, the **struct procsinfo64** and *sizeof* (**struct procsinfo64**) can be used by 32-bit **getprocs** to return full 64-bit process information. Note that the **procsinfo** structure not only increases certain **procsinfo** fields from 32 to 64 bits, but that it contains additional information not present in **procsinfo**. The **struct procsinfo64** contains the same data as **struct procsinfo** when compiled in a 64-bit program.

The 64-bit applications are required to use **getprocs64()** and **procentry64**. Note that **struct procentry64** contains the same information as **struct procsinfo64**, with the addition of support for the 64-bit time_t and dev_t, and the 256-bit sigset_t. The **procentry64** structure also contains a new version of **struct ucred** (**struct ucred_ext**) and a new, expanded **struct rusage** (**struct trusage64**) as described in **<sys/cred.h>** and **<sys/resource.h>** respectively. Application developers are also encouraged to use **getprocs64()** in 32-bit applications to obtain 64-bit process information as this interface provides the new, larger types. The **getprocs()** interface will still be supported for 32-bit applications using **struct procsinfo** or **struct procsinfo64** but will not be available to 64-bit applications.

Parameters

ProcessBuffer
Specifies the starting address of an array of **procsinfo**, **procsinfo64**, or **procentry64** structures to be filled in with process table entries. If a value of **NULL** is passed for this parameter, the **getprocs** subroutine scans the process table and sets return values as normal, but no process entries are retrieved.

Note: The *ProcessBuffer* parameter of **getprocs** subroutine contains two struct rusage fields named **pi_ru** and **pi_cru**. Each of these fields contains two struct timeval fields named **ru_utime** and **ru_stime**. The **tv_usec** field in both of the struct timeval contain nanoseconds instead of microseconds. These values cone from the struct user fields named **U_ru** and **U_cru**. The **pi_cru_*** fields also contain the page faults for reaped child which roll back to parent. This field is updated before the child can become zombie.

ProcessSize
Specifies the size of a single **procsinfo**, **procsinfo64**, or **procentry64** structure.

FileBuffer
Specifies the starting address of an array of **fdsinfo**, or **fdsinfo64** structures to be filled in with per-process file descriptor information. If a value of **NULL** is passed for this parameter, the **getprocs** subroutine scans the process table and sets return values as normal, but no file descriptor entries are retrieved.

Note: Use **fdsinfo64_100K** when processes have more than 32 K file descriptors.

FileSize
Specifies the size of a single **fdsinfo**, or **fdsinfo64** structure.

Note: Use **fdsinfo64_100K** when processes have more than 32 K file descriptors.

IndexPointer
Specifies the address of a process identifier which indicates the required process table entry. A process identifier of zero selects the first entry in the table. The process identifier is updated to indicate the next entry to be retrieved.

Note: The *IndexPointer* does not have to correspond to an existing process, and may in fact correspond to a different process than the one you expect. There is no guarantee that the process slot pointed to by *IndexPointer* will contain the same process between successive calls to **getprocs()** or **getprocs64()**.

Count
Specifies the number of process table entries requested.

Return Values

If successful, the **getprocs** subroutine returns the number of process table entries retrieved; if this is less than the number requested, the end of the process table has been reached. A value of 0 is returned when the end of the process table has been reached. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **getprocs** subroutine does not succeed if the following are true:

Item	Description
EINVAL	The *ProcessSize* or *FileSize* parameters are invalid, or the *IndexPointer* parameter does not point to a valid process identifier, or the *Count* parameter is not greater than zero.
EFAULT	The copy operation to one of the buffers was not successful.

getproj Subroutine

Purpose

Retrieves the project definition from the kernel project registry for the requested project name.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

getproj(struct project *, int flag)
```

Description

The **getproj** subroutine functions similar to the **getprojs** subroutine with the exception that the **getproj** subroutine retrieves the definition only for the project name or number, which is passed as its argument. The *flag* parameter indicates what is passed. The *flag* parameter has the following values:

- PROJ_NAME — Indicates that the supplied project definition only has the project name. The **getproj** subroutine queries the kernel to obtain a match for the supplied project name and returns the matching entry.
- PROJ_NUM — Indicates that the supplied project definition only has the project number. The **getproj** subroutine queries the kernel to obtain a match for the supplied project number and returns the matching entry.

Generally, the projects are loaded from the system project definition file or LDAP, or from both. When more than one of these project repositories are used, project name and project ID collisions are possible. These projects are differentiated by the kernel using an origin flag. This origin flag designates the project repository from where the project definition is obtained. If the caller wants to retrieve the project definition that belongs to a specific project repository, the specific origin value should be passed in the flags field of the project structure. Valid project origins values that can be passed are defined in the `sys/aacct.h` file. If the projects are currently loaded from the project repository represented by the origin value, `getproj` returns the specified project if it exists. If the origin value is not passed, the first project reference found in the kernel registry is returned. Regardless of whether the origin is passed or not, `getproj` always returns the project origin flags in the output project structure.

Parameters

Item	Description
project	Pointer holding the project whose information is required.
flag	An integer flag that indicates whether the match needs to be performed on the supplied project name or number.

Security

There are no restrictions. Any user can call this function.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Invalid argument. The *flag* parameter is not valid or the passed pointer is NULL.
ENOENT	Project not found.

getprojdb Subroutine

Purpose

Retrieves the specified project record from the project database.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

getprojdb(void *handle, struct project *project,  int flag)
```

Description

The **getprojdb** subroutine searches the project database associated with the *handle* parameter for the specified project. The project database must be initialized before calling this subroutine. The routines **projdballoc** and **projdbfinit** are provided for this purpose. The *flag* parameter indicates the type of search. The following flags are defined:

- **PROJ_NAME** — Search by product name. The **getprojdb** subroutine scans the file to obtain a match for the supplied project name and returns the matching entry.
- **PROJ_NUM** — Search by product number. The **getprojdb** subroutine scans the file to obtain a match for the supplied project number and returns the matching entry.

The entire database is searched. If the specified record is found, the **getprojdb** subroutine stores the relevant project information into the `struct project` buffer, which is passed as an argument to this subroutine. The specified project is then made the current project in the database. If the specified project is not found, the database is reset so that the first project in the database is the current project.

Parameters

Item	Description
handle	Pointer to the handle allocated for the project database.
project	Pointer holding the project name whose information is required.

Item	Description
flag	Integer flag indicating what type of information is sent for matching; that is, whether the match needs to be performed by project name or number.

Security

No restrictions. Any user can call this function.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
ENOENT	Project definition not found.
EINVAL	Invalid arguments if flag is not valid or passed pointer is NULL.

getprojs Subroutine

Purpose

Retrieves the project details from the kernel project registry.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

getprojs(struct project *, int *)
```

Description

The **getprojs** subroutine retrieves the specified number of project definitions from the kernel project registry. The number of definitions to be retrieved is passed as an argument to this subroutine, and it is also passed with a buffer of type **struct *project***, where the retrieved project definitions are stored.

When the **getprojs** subroutine is called with a NULL value passed instead of a pointer to a **struct *project***, the **getprojs** subroutine returns the total number of defined projects in the kernel project registry. This number can be used by any subsequent calls to retrieve the project details.

If the integer value passed is smaller than the number of project definitions available, then the project buffer will be filled with as many entries as requested. If the value is greater than the number of available definitions, then the available records are filled in the structure and the integer value is updated with the number of records actually retrieved.

Generally, the projects are loaded from the system project definition file or LDAP, or from both. When more than one of these project repositories are used, project name and project ID collisions are possible. These projects are differentiated by the kernel using an origin flag. This origin flag designates the project repository from where the project definition is obtained. Valid project origins values that can be passed

are defined in the `sys/aacct.h` file. The `getproj` subroutine also returns this origin information in the `flags` field of the output project structures.

Parameters

Item	Description
pointer	Points to a project structure where the retrieved data is stored.
int	An integer that indicates the number of elements to be retrieved.

Security

There are no restrictions. Any user can call this function.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Invalid arguments if passed *int* pointer is NULL
ENOENT	No projects available.

getpw Subroutine

Purpose

Retrieves a user's /etc/passwd file entry.

Library

Standard C Library **(libc.a)**

Syntax

int getpw (*UserID*, *Buffer*)

```
uid_t UserID
char *Buffer
```

Description

The **getpw** subroutine opens the **/etc/passwd** file and returns, in the *Buffer* parameter, the **/etc/passwd** file entry of the user specified by the *UserID* parameter.

Parameters

Item	Description
Buffer	Specifies a character buffer large enough to hold any **/etc/passwd** entry.
UserID	Specifies the ID of the user for which the entry is desired.

Return Values

The **getpw** subroutine returns:

Item	Description
0	Successful completion
-1	Not successful.

getpwent, getpwuid, getpwnam, putpwent, setpwent, or endpwent Subroutine

Purpose

Accesses the basic user information in the user database.

Library

Standard C Library **(libc.a)**

Syntax

```
#include <sys/types.h>
#include <pwd.h>
```

```
struct passwd *getpwent ( )
```

```
struct passwd *getpwuid ( UserID)
uid_t UserID;
```

```
struct passwd *getpwnam ( Name)
char *Name;
```

```
int putpwent ( Password,  File)
struct passwd *Password;
FILE *File;
```

```
void setpwent ( )
```

```
void endpwent ( )
```

Description

Attention: All information generated by the **getpwent**, **getpwnam**, and **getpwuid** subroutines is stored in a static area. Subsequent calls to these subroutines overwrite this static area. To save the information in the static area, applications should copy it.

Attention: The **getpwent** subroutine is only supported by LOCAL and NIS load modules, not any other LAM authentication module.

These subroutines access the basic user attributes.

The **setpwent** subroutine opens the user database if it is not already open. Then, this subroutine sets the cursor to point to the first user entry in the database. The **endpwent** subroutine closes the user database.

The **getpwent**, **getpwnam**, and **getpwuid** subroutines return information about a user. These subroutines do the following:

Item	Description
getpwent	Returns the next user entry in the sequential search.
getpwnam	Returns the first user entry in the database whose name matches the *Name* parameter.
getpwuid	Returns the first user entry in the database whose ID matches the *UserID* parameter.

The **putpwent** subroutine writes a password entry into a file in the colon-separated format of the **/etc/passwd** file.

The passwd structure

The **getpwent**, **getpwnam**, and **getpwuid** subroutines return a **passwd** structure. The **passwd** structure is defined in the **pwd.h** file and has the following fields:

Item	Description
pw_name	Contains the name of the user name.
pw_passwd	Contains the user's encrypted password.
	Note: If the password is not stored in the **/etc/passwd** file and the invoker does not have access to the shadow file that contains passwords, this field contains an undecryptable string, usually an * (asterisk).
pw_uid	Contains the user's ID.
pw_gid	Identifies the user's principal group ID.
pw_gecos	Contains general user information.
pw_dir	Identifies the user's home directory.
pw_shell	Identifies the user's login shell.

Note: If Network Information Services (NIS) is enabled on the system, these subroutines attempt to retrieve the information from the NIS authentication server before attempting to retrieve the information locally.

Parameters

Item	Description
File	Points to an open file whose format is similar to the **/etc/passwd** file format.
Name	Specifies the user name.
Password	Points to a password structure. This structure contains user attributes.
UserID	Specifies the user ID.

Security

Item	Description
Files Accessed:	

Mode	File
rw	**/etc/passwd** (write access for the **putpwent** subroutine only)
r	**/etc/security/passwd** (if the password is desired)

Return values

The **getpwent**, **getpwnam**, and **getpwuid** subroutines return a pointer to a valid password structure if successful. Otherwise, a null pointer is returned.

The **getpwent** subroutine will return a null pointer and an **errno** value of **ENOATTR** when it detects a corrupt entry. To get subsequent entries following the corrupt entry, call the **getpwent** subroutine again.

Error codes

If any of these subroutines fail, the following errors might be returned:

Item	Description
EACCES	Access permission is denied for the data request.
EINVAL	The parameter does not contain one of the defined attributes or null.
EIO	Failed to access remote user database.
ENOATTR	The specified attribute is not defined for the current user.
EPERM	Operation is not permitted.
ESRCH	The specified parameter does not exist.

Files

Item	Description
/etc/passwd	Contains user IDs and their passwords

getrlimit, getrlimit64, setrlimit, setrlimit64, or vlimit Subroutine

Purpose

Controls maximum system resource consumption.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>
#include <sys/resource.h>

int setrlimit( Resource1, RLP)
int Resource1;
struct rlimit *RLP;

int setrlimit64 ( Resource1, RLP)
int Resource1;
struct rlimit64 *RLP;

int getrlimit ( Resource1, RLP)
int Resource1;
struct rlimit *RLP;
```

```
int getrlimit64 ( Resource1, RLP)
int Resource1;
struct rlimit64 *RLP;

#include <sys/vlimit.h>

vlimit ( Resource2, Value)
int Resource2, Value;
```

Description

The **getrlimit** subroutine returns the values of limits on system resources used by the current process and its children processes. The **setrlimit** subroutine sets these limits. The **vlimit** subroutine is also supported, but the **getrlimit** subroutine replaces it.

A resource limit is specified as either a soft (current) or hard limit. A calling process can raise or lower its own soft limits, but it cannot raise its soft limits above its hard limits. A calling process must have root user authority to raise a hard limit.

Note: The initial values returned by the **getrlimit** subroutine are the **ulimit** values in effect when the process was started. For *maxdata* programs the initial value returned by **getrlimit** for the soft data limit is the lower of the hard data limit or the *maxdata* value. When a program is executing using the large address-space model, the operating system attempts to modify the soft limit on data size, if necessary, to increase it to match the *maxdata* value. If the *maxdata* value is larger than the current hard limit on data size, either the program will not execute if the *XPG_SUS_ENV* environment variable has the value set to ON, or the soft limit will be set to the current hard limit. If the *maxdata* value is smaller than the size of the program's static data, the program will not execute.

The **rlimit** structure specifies the hard and soft limits for a resource, as defined in the **sys/resource.h** file. The **RLIM_INFINITY** value defines an infinite value for a limit.

When compiled in 32-bit mode, RLIM_INFINITY is a 32-bit value; when compiled in 64-bit mode, it is a 64-bit value. 32-bit routines should use **RLIM64_INFINITY** when setting 64-bit limits with the **setrlimit64** routine, and recognize this value when returned by **getrlimit64**.

This information is stored as per-process information. This subroutine must be executed directly by the shell if it is to affect all future processes created by the shell.

Note: Raising the data limit does not raise the program break value. Use the **brk/sbrk** subroutines to raise the break value. If the proper memory segments are not initialized at program load time, raising your memory limit will not allow access to this memory. Use the **-bmaxdata** flag of the **ld** command to set up these segments at load time.

When compiled in 32-bit mode, the **struct rlimit** values may be returned as RLIM_SAVED_MAX or RLIM_SAVED_CUR when the actual resource limit is too large to represent as a 32-bit **rlim_t**.

These values can be used by library routines which set their own **rlimits** to save off potentially 64-bit **rlimit** values (and prevent them from being truncated by the 32-bit **struct rlimit**). Unless the library routine intends to permanently change the **rlimits**, the RLIM_SAVED_MAX and RLIM_SAVED_CUR values can be used to restore the 64-bit **rlimits**.

Application limits may be further constrained by available memory or implementation defined constants such as **OPEN_MAX** (maximum available open files).

Parameters

Item	Description
Resource1	Can be one of the following values: **RLIMIT_AS** The maximum size, in bytes, of the total available memory of a process. This limit is enforced by the kernel only if the *XPG_SUS_ENV=ON* environment variable is set in the user's environment before the process is executed. If the *XPG_SUS_ENV* environment variable is not set in the user's environment, the limit is not enforced. **RLIMIT_CORE** The largest size, in bytes, of a **core** file that can be created. This limit is enforced by the kernel. If the value of the **RLIMIT_FSIZE** limit is less than the value of the **RLIMIT_CORE** limit, the system uses the **RLIMIT_FSIZE** limit value as the soft limit. **RLIMIT_CPU** The maximum amount of central processing unit (processor) time, in seconds, to be used by each process. If a process exceeds its soft processor limit, the kernel will send a **SIGXCPU** signal to the process. After the hard limit is reached, the process will be killed with **SIGXCPU**, even if it handles, blocks, or ignores that signal. **RLIMIT_DATA** The maximum size, in bytes, of the data region for a process. This limit defines how far a program can extend its break value with the **sbrk** subroutine. This limit is enforced by the kernel. If the *XPG_SUS_ENV=ON* environment variable is set in the user's environment before the process is executed and a process attempts to set the limit lower than current usage, the operation fails with the value of *errno* global variable set to **EINVAL**. If the *XPG_SUS_ENV* environment variable is not set, the operation fails with the value of *errno* global variable set to **EFAULT**. **RLIMIT_FSIZE** The largest size, in bytes, of any single file that can be created. When a process attempts to write, truncate, or clear beyond its soft **RLIMIT_FSIZE** limit, the operation will fail with the value of *errno* global variable set to **EFBIG**. If the *XPG_SUS_ENV=ON* environment variable is set in the user's environment before the process is executed, the **SIGXFSZ** signal is also generated. **RLIMIT_NOFILE** This is a number one greater than the maximum value that the system may assign to a newly-created descriptor. **RLIMIT_STACK** The maximum size, in bytes, of the stack region for a process. This limit defines how far a program stack region can be extended. Stack extension is performed automatically by the system. This limit is enforced by the kernel. When the stack limit is reached, the process receives a **SIGSEGV** signal. If this signal is not caught by a handler by using the signal stack, the signal ends the process. **RLIMIT_RSS** The maximum size, in bytes, to which the resident set size of a process can grow. This limit is not enforced by the kernel. A process may exceed its soft limit size without being ended. **RLIMIT_THREADS** The maximum number of threads each process can create. This limit is enforced by the kernel and the pthread debug library. **RLIMIT_NPROC** The maximum number of processes each user can create.
RLP	Points to the **rlimit** or **rlimit64** structure, which contains the soft (current) and hard limits. For the **getrlimit** subroutine, the requested limits are returned in this structure. For the **setrlimit** subroutine, the desired new limits are specified here.
Resource2	The flags for this parameter are defined in the **sys/vlimit.h**, and are mapped to corresponding flags for the **setrlimit** subroutine.
Value	Specifies an integer used as a soft-limit parameter to the **vlimit** subroutine.

Return Values

On successful completion, a return value of 0 is returned, changing or returning the resource limit. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error. If the current limit specified is beyond the hard limit, the **setrlimit** subroutine sets the limit to max limit and returns successfully.

Error Codes

The **getrlimit**, **getrlimit64**, **setrlimit**, **setrlimit64**, or **vlimit** subroutine is unsuccessful if one of the following is true:

Item	Description
EFAULT	The address specified for the *RLP* parameter is not valid.

Item	Description
EINVAL	The *Resource1* parameter is not a valid resource, or the limit specified in the *RLP* parameter is invalid.
EPERM	The limit specified to the **setrlimit** subroutine would have raised the maximum limit value, and the caller does not have root user authority.

getrpcent, getrpcbyname, getrpcbynumber, setrpcent, or endrpcent Subroutine

Purpose

Accesses the **/etc/rpc** file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <netdb.h>

struct rpcent *getrpcent ()
struct rpcent *getrpcbyname ( Name)
char *Name;
struct rpcent *getrpcbynumber ( Number)
int Number;
void setrpcent (StayOpen)
int  StayOpen
void endrpcent
```

Description

Attention: Do not use the **getrpcent**, **getrpcbyname**, **getrpcbynumber**, **setrpcent**, or **endrpcent** subroutine in a multithreaded environment.

Attention: The information returned by the **getrpcbyname**, and **getrpcbynumber** subroutines is stored in a static area and is overwritten on subsequent calls. Copy the information to save it.

The **getprcbyname** and **getrpcbynumber** subroutines each return a pointer to an object with the **rpcent** structure. This structure contains the broken-out fields of a line from the **/etc/rpc** file. The **getprcbyname** and **getrpcbynumber** subroutines searches the **rpc** file sequentially from the beginning of the file until it finds a matching RPC program name or number, or until it reaches the end of the file. The **getrpcent** subroutine reads the next line of the file, opening the file if necessary.

The **setrpcent** subroutine opens and rewinds the **/etc/rpc** file. If the *StayOpen* parameter does not equal 0, the **rpc** file is not closed after a call to the **getrpcent** subroutine.

The **setrpcent** subroutine rewinds the **rpc** file. The **endrpcent** subroutine closes it.

The **rpc** file contains information about Remote Procedure Call (RPC) programs. The **rpcent** structure is in the **/usr/include/netdb.h** file and contains the following fields:

Item	Description
`r_name`	Contains the name of the server for an RPC program
`r_aliases`	Contains an alternate list of names for RPC programs. This list ends with a 0.
`r_number`	Contains a number associated with an RPC program.

Parameters

Item	Description
Name	Specifies the name of a server for **rpc** program.
Number	Specifies the **rpc** program number for service.
StayOpen	Contains a value used to indicate whether to close the **rpc** file.

Return Values

These subroutines return a null pointer when they encounter the end of a file or an error.

Files

Item	Description
/etc/rpc	Contains information about Remote Procedure Call (RPC) programs.

getrusage, getrusage64, times, or vtimes Subroutine

Purpose

Displays information about resource use.

Libraries

getrusage, **getrusage64**, **times**: Standard C Library (**libc.a**)

Item	Description
vtimes:	Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <sys/times.h>
#include <sys/resource.h>
```

```
int getrusage ( Who,  RUsage)
int Who;
struct rusage *RUsage;
```

```
int getrusage64 ( Who,  RUsage)
int Who;
struct rusage64 *RUsage;
```

```
#include <sys/types.h>
#include <sys/times.h>
```

```
clock_t times ( Buffer)
struct tms *Buffer;
```

```
#include <sys/times.h>
```

```
vtimes ( ParentVM,  ChildVM)
struct vtimes *ParentVm, ChildVm;
```

Description

The **getrusage** subroutine displays information about how resources are used by the current process or all completed child processes.

When compiled in 64-bit mode, **rusage** counters are 64 bits. If **getrusage** is compiled in 32-bit mode, **rusage** counters are 32 bits. If the kernel's value of a **usage** counter has exceeded the capacity of the corresponding 32-bit **rusage** value being returned, the **rusage** value is set to INT_MAX.

The **getrusage64** subroutine can be called to make 64-bit **rusage** counters explicitly available in a 32-bit environment.

64-bit quantities are also available to 64-bit applications through the **getrusage()** interface in the ru_utime and ru_stime fields of **struct rusage**.

The **times** subroutine fills the structure pointed to by the *Buffer* parameter with time-accounting information. All time values reported by the **times** subroutine are measured in terms of the number of clock ticks used. Applications should use **sysconf** (**_SC_CLK_TCK**) to determine the number of clock ticks per second.

The **tms** structure defined in the **/usr/include/sys/times.h** file contains the following fields:

```
time_t   tms_utime;
```

```
time_t   tms_stime;
```

```
time_t   tms_cutime;
```

```
time_t   tms_cstime;
```

This information is read from the calling process as well as from each completed child process for which the calling process executed a **wait** subroutine.

Item	Description
`tms_utime`	The CPU time used for executing instructions in the user space of the calling process
`tms_stime`	The CPU time used by the system on behalf of the calling process.
`tms_cutime`	The sum of the `tms_utime` and the `tms_cutime` values for all the child processes.
`tms_cstime`	The sum of the `tms_stime` and the `tms_cstime` values for all the child processes.

Note: The system measures time by counting clock interrupts. The precision of the values reported by the **times** subroutine depends on the rate at which the clock interrupts occur.

The **vtimes** subroutine is supported to provide compatibility with earlier programs.

The **vtimes** subroutine returns accounting information for the current process and for the completed child processes of the current process. Either the *ParentVm* parameter, the *ChildVm* parameter, or both may be 0. In that case, only the information for the nonzero pointers is returned.

After a call to the **vtimes** subroutine, each buffer contains information as defined by the contents of the **/usr/include/sys/vtimes.h** file.

Parameters

Item	Description
Who	Specifies a value of **RUSAGE_THREAD**, **RUSAGE_SELF**, or **RUSAGE_CHILDREN**.

Item	Description
RUsage	Points to a buffer described in the **/usr/include/sys/resource.h** file. The fields are interpreted as follows: **ru_utime** The total amount of time running in user mode. **ru_stime** The total amount of time spent in the system executing on behalf of the processes. **ru_maxrss** The maximum size, in kilobytes, of the used resident set size. **ru_ixrss** An integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in units of kilobytes * seconds-of-execution and is calculated by adding the number of shared memory pages in use each time the internal system clock ticks, and then averaging over one-second intervals. **ru_idrss** An integral value of the amount of unshared memory in the data segment of a process (expressed in units of kilobytes * seconds-of-execution). **ru_minflt** The number of page faults serviced without any I/O activity. In this case, I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation. **ru_majflt** The number of page faults serviced that required I/O activity. **ru_nswap** The number of times a process was swapped out of main memory. **ru_inblock** The number of times the file system performed input. **ru_oublock** The number of times the file system performed output. **Note:** The numbers that the `ru_inblock` and `ru_oublock` fields display account for real I/O only; data supplied by the caching mechanism is charged only to the first process to read or write the data. **ru_msgsnd** The number of IPC messages sent. **ru_msgrcv** The number of IPC messages received. **ru_nsignals** The number of signals delivered. **ru_nvcsw** The number of times a context switch resulted because a process voluntarily gave up the processor before its time slice was completed. This usually occurs while the process waits for availability of a resource. **ru_nivcsw** The number of times a context switch resulted because a higher priority process ran or because the current process exceeded its time slice.
Buffer	Points to a **tms** structure.
ParentVm	Points to a **vtimes** structure that contains the accounting information for the current process.

Item	Description
ChildVm	Points to a **vtimes** structure that contains the accounting information for the terminated child processes of the current process.

Return Values

Upon successful completion, the **getrusage** and **getrusage64** subroutines return a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Upon successful completion, the **times** subroutine returns the elapsed real time in units of ticks, whether profiling is enabled or disabled. This reference time does not change from one call of the **times** subroutine to another. If the **times** subroutine fails, it returns a value of -1 and sets the **errno** global variable to indicate the error.

Error Codes

The **getrusage** and **getrusage64** subroutines do not run successfully if either of the following is true:

Item	Description
EINVAL	The *Who* parameter is not a valid value.
EFAULT	The address specified for *RUsage* is not valid.

The **times** subroutine does not run successfully if the following is true:

Item	Description
EFAULT	The address specified by the *buffer* parameter is not valid.

getroleattr, nextrole or putroleattr Subroutine

Purpose

Accesses the role information in the roles database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int getroleattr(Role, Attribute, Value, Type)
char *Role;
char *Attribute;
void *Value;
int Type;
```

```
char *nextrole(void)
```

```
int putroleattr(Role, Attribute, Value, Type)
char *Role;
char *Attribute;
void *Value;
int Type;
```

Description

The **getroleattr** subroutine reads a specified attribute from the role database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the **putroleattr** subroutine writes a specified attribute into the role database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putroleattr** subroutine must be explicitly committed by calling the **putroleattr** subroutine with a Type parameter specifying SEC_COMMIT. Until all the data is committed, only the **getroleattr** subroutine within the process returns written data.

The **nextrole** subroutine returns the next role in a linear search of the role database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setroledb** and **endroledb** subroutines should be used to open and close the role database.

Parameters

Item	Description
Attribute	Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file: **S_AUDITCLASSES** Audit classes to which the role belongs. The attribute type is **SEC_LIST**. **S_ROLELIST** List of roles included by this role. The attribute type is **SEC_LIST**. **S_AUTHORIZATIONS** List of authorizations included by this role. The attribute type is **SEC_LIST**. **S_GROUPS** List of groups required for this role. The attribute type is **SEC_LIST**. **S_HOSTSENABLEDROLE** List of hosts from where the role can be downloaded to the Kernel Role Table. The attribute type is **SEC_LIST**. **S_HOSTSDISABLEDROLE** List of hosts from where the role cannot be downloaded to the Kernel Role Table. The attribute type is **SEC_LIST**. **S_SCREENS** List of SMIT screens required for this role. The attribute type is **SEC_LIST**. **S_VISIBILITY** Number value stating the visibility of the role. The attribute type is **SEC_INT**. **S_MSGCAT** Message catalog file name. The attribute type is **SEC_CHAR**. **S_MSGNUMBER** Message number within the catalog. The attribute type is **SEC_INT**. **S_MSGSET** Message catalog set number. The attribute type is **SEC_INT**. **S_ID** Role identifier. The attribute type is **SEC_INT**. **S_DFLTMSG** Default role description string used when catalogs are not in use. The attribute type is **SEC_CHAR**. **S_USERS** List of users that have been assigned this role. This attribute is a read only attribute and cannot be modified through the **putroleattr** subroutine. The attribute type is **SEC_LIST**. **S_AUTH_MODE** The authentication to use when assuming the role through the **swrole** command. Valid values are **NONE** and **INVOKER**. The attribute type is **SEC_CHAR**.

Item	Description
Type	Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include: **SEC_INT** The format of the attribute is an integer. For the **getroleattr** subroutine, the user should supply a pointer to a defined integer variable. For the **putroleattr** subroutine, the user should supply an integer. **SEC_CHAR** The format of the attribute is a null-terminated character string. For the **getroleattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putroleattr** subroutine, the user should supply a character pointer. **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string. For the **getroleattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putroleattr** subroutine, the user should supply a character pointer. **SEC_COMMIT** For the **putroleattr** subroutine, this value specified by itself indicates that changes to the named role are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no role is specified, the changes to all modified roles are committed to permanent storage. **SEC_DELETE** The corresponding attribute is deleted from the database. **SEC_NEW** Updates the role database file with the new role name when using the **putroleattr** subroutine.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Return Values

If successful, the **getroleattr** returns 0. Otherwise, a value of -1 is returned and the **errno** global variables is set to indicate the error.

Error Codes

Possible return codes are:

Item	Description
EACCES	Access permission is denied for the data request.
ENOENT	The specified *Role* parameter does not exist.
ENOATTR	The specified role attribute does not exist for this role.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or null.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.

Item	Description
EPERM	Operation is not permitted.

getroleattrs Subroutine

Purpose

Retrieves multiple role attributes from the role database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getroleattrs(Role, Attributes, Count)
    char *Role;
    dbattr_t *Attributes;
    int Count;
```

Description

The **getroleattrs** reads one or more attributes from the role database. The role specified with the *Role* parameter must already exist in the role database. The *Attributes* parameter contains information about each attribute that is to be read. All attributes specified by the *Attributes* parameter must be examined on a successful call to the **getroleattrs** subroutine to determine whether value of the *Attributes* parameter was successfully retrieved. Attributes of the **SEC_CHAR** or **SEC_LIST** type will have their values returned to the allocated memory. Caller need to free this memory. The **dbattr_t data** structure contains the following fields:

Item	Description
attr_name	The name of the target role attribute.
attr_idx	This attribute is used internally by the **getroleattrs** subroutine.
attr_type	The type of the target attribute.
attr _flag	The result of the request to read the target attribute. On successful completion, the value of zero is returned. Otherwise, it returns a value of nonzero.
attr_un	A union that contains the returned values for the requested query.
attr_domain	The subroutine ignores any input to this field. If this field is set to null, the subroutine sets this field to the name of the domain where the role is found.

The following valid role attributes for the **getroleattrs** subroutine are defined in the **usersec.h** file:

Name	Description	Type
S_AUDITCLASSES	Audit classes to which the role belongs.	**SEC_LIST**
S_AUTHORIZATIONS	Retrieves all the authorizations that are assigned to the role.	**SEC_LIST**

Name	Description	Type
S_AUTH_MODE	The authentication to perform when assuming the role through the **swrole** command. It contains the following possible values: **NONE** No authentication is required. **INVOKER** This is the default value. Invokers of the **swrole** command must enter their passwords to assume the role.	**SEC_CHAR**
S_DFLTMSG	The default role description that is used when catalogs are not in use.	**SEC_CHAR**
S_GROUPS	The groups that a user is suggested to become a member of. It is for informational purpose only.	**SEC_LIST**
S_HOSTSENABLEDROLE	The list of hosts from where the role can be downloaded to the Kernel Role Table.	**SEC_LIST**
S_HOSTSDISABLEDROLE	The list of hosts from where the role cannot be downloaded to the Kernel Role Table.	**SEC_LIST**
S_ID	The role identifier.	**SEC_INT**
S_MSGCAT	The message catalog name that contains the role description.	**SEC_CHAR**
S_MSGSET	The message catalog's set number for the role description.	**SEC_INT**
S_MSGNUMBER	The message number for the role description.	**SEC_INT**
S_ROLELIST	Lists of roles whose authorizations are included in this role.	**SEC_LIST**
S_ROLES	Retrieves all the roles that are available on the system. It is valid only when the *Role* parameter is set to **ALL**.	**SEC_LIST**
S_SCREENS	The SMIT screens that the role can access.	**SEC_LIST**

Name	Description	Type
S_VISIBILITY	An integer that determines whether the role is active or not. It contains the following possible values: **-1** The role is disabled. **0** The role is active but not visible from a GUI. **1** The role is active and visible. This is the default value.	**SEC_INT**
S_USERS	Lists of users that have been assigned this role.	**SEC_LIST**

The following union members correspond to the definitions of the **attr_char**, **attr_int**, **attr_long** and the **attr_llong** macros in the **usersec.h** file respectively.

Item	Description
au_char	The attributes of the **SEC_CHAR** and **SEC_LIST** types store a pointer to the returned value in this member when the attributes are successfully retrieved. The caller is responsible for freeing this memory.
au_int	The storage location for attributes of the **SEC_INT** type.
au_long	The storage location for attributes of the **SEC_LONG** type.
au_llong	The storage location for attributes of the **SEC_LLONG** type.

If **ALL** is specified for the *Role* parameter, the only valid attribute that can be displayed in the *Attribute* parameter is the **S_ROLES** attribute. Specifying any other attribute with a role name of **ALL** causes the **getroleattrs** subroutine to fail.

Parameters

Item	Description
Role	Specifies the role name for which the attributes are to be read.
Attributes	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of role attributes is defined in the **usersec.h** header file.
Count	The number of attributes specified in the *Attributes* parameter.

Security

Files Accessed:

File	Mode
/etc/security/roles	r

Return Values

If the role specified by the *Role* parameter exists in the role database, the **getroleattrs** subroutine returns zero. On successful completion, the **attr_flag** attribute of each attribute that is specified in the *Attributes* parameter must be examined to determine whether it was successfully retrieved. If the specified role does not exist, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **getroleattrs** subroutine returns -1, one of the following **errno** values is set:

Item	Description
EINVAL	The *Role* parameter is **NULL**.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Role* parameter is **NULL** and the *Count* parameter is greater than zero.
EINVAL	The *Role* parameter is **ALL** but the *Attributes* parameter contains an attribute other than **S_ROLES**.
ENOENT	The role specified in the *Role* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

If the **getroleattrs** subroutine fails to query an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding value of the *Attributes* parameter:

Item	Description
EACCES	The invoker does not have access to the attribute specified in the **attr_name** field.
EINVAL	The **attr_name** field in the *Attributes* parameter is not a recognized role attribute.
EINVAL	The **attr_type** field in the *Attributes* parameter contains a type that is not valid.
EINVAL	The **attr_un** field in the *Attributes* parameter does not point to a valid buffer.
ENOATTR	The **attr_name** field in the *Attributes* parameter specifies a valid attribute, but no value is defined for this role.

gets or fgets Subroutine

Purpose

Gets a string from a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>
char *gets ( String)
char *String;
```

```
char *fgets (String,  Number,  Stream)
char *String;
int Number;
FILE *Stream;
```

Description

The **gets** subroutine reads bytes from the standard input stream, **stdin**, into the array pointed to by the *String* parameter. It reads data until it reaches a new-line character or an end-of-file condition. If a new-line character stops the reading process, the **gets** subroutine discards the new-line character and terminates the string with a null character.

The **fgets** subroutine reads bytes from the data pointed to by the *Stream* parameter into the array pointed to by the *String* parameter. The **fgets** subroutine reads data up to the number of bytes specified by the *Number* parameter minus 1, or until it reads a new-line character and transfers that character to the *String* parameter, or until it encounters an end-of-file condition. The **fgets** subroutine then terminates the data string with a null character.

The first successful run of the **fgetc**, **fgets**, **fgetwc**, **fgetws**, **fread**, **fscanf**, **getc**, **getchar**, **gets** or **scanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc**or **ungetwc** subroutine marks the `st_atime` field for update.

Parameters

Item	Description
String	Points to a string to receive bytes.
Stream	Points to the **FILE** structure of an open file.
Number	Specifies the upper bound on the number of bytes to read.

Return Values

If the **gets** or **fgets** subroutine encounters the end of the file without reading any bytes, it transfers no bytes to the *String* parameter and returns a null pointer. If a read error occurs, the **gets** or **fgets** subroutine returns a null pointer and sets the **errno** global variable (errors are the same as for the **fgetc** subroutine). Otherwise, the **gets** or **fgets** subroutine returns the value of the *String* parameter.

> **Note:** Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding the **SA_RESTART** value.

getsecconfig and setsecconfig Subroutines

Purpose

Retrieves and sets the kernel security configuration flags for system run mode.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>

int getsecconfig (secconf)
uint32_t *secconf;

int setsecconfig(secconf, mode)
```

```
uint32_t secconf;
ushort mode;
```

Description

The **getsecconfig** subroutine retrieves the security configuration flags based on the current run mode. The flags are copied to kernel security configuration flag specified by the *secconf* parameter.

The **setsecconfig** subroutine sets the kernel security configuration for the specified mode according to flag that the *secconf* parameter specifies. The kernel configuration flags can only be changed in the CONFIGURATION runtime mode.

Parameters

Item	Description
secconf	Specifies the kernel security configuration flags.
Mode	Specifies the runtime mode to be updated. The valid values are CONFIGURATION_MODE and OPERATIONAL_MODE.

Security

Access Control: To set the configuration flags, the calling process invoking should have the PV_KER_SECCONFIG privilege.

Return Values

If successful, these subroutines return a value of zero. Otherwise, they return a value of -1.

Error Codes

If these subroutines fail, they set one of the following error codes:

Item	Description
EINVAL	The value that the parameter specifies is null.
EINVAL	The specified run time mode is not valid.
EINVAL	The configuration flags that are specified are not proper.
EPERM	The calling process either does not have permissions or privileges, or the system is not in the CONFIGURATION runtime mode.

getsecorder Subroutine

Purpose

Retrieves the ordering of domains for certain security databases.

Library

Standard C Library (**libc.a**)

Syntax

```
char * getsecorder (name)
       char *name;
```

Description

The **getsecorder** subroutine returns the value of the domain order for the database specified by the *name* parameter. When a previous call to the **setsecorder** subroutine with a valid value is successful, the **getsecorder** subroutine returns that value. Otherwise, the value of the **secorder** attribute of the name database in the **/etc/nscontrol.conf** file is returned. The returned value is a comma separated list of module names. The caller must free it after use. This subroutine is thread safe.

Parameters

Item	Description
name	Specifies the database name. The parameter can have one of the following valid values: • **authorizations** • **roles** • **privcmds** • **privdevs** • **privfiles**

Security

Files Accessed:

File	Mode
/etc/nscontrol.conf	r

Return Values

On successful completion, a comma-separated list of module names is returned. If the subroutine fails, it returns a value of NULL and sets the **errno** value to indicate the error.

Error Codes

Item	Description
EINVAL	The database name is not valid.
ENOMEM	Unable to allocate memory.

getfsent_r, getfsspec_r, getfsfile_r, getfstype_r, setfsent_r, or endfsent_r Subroutine

Purpose

Gets information about a file system.

Library

Thread-Safe C Library (**libc_r.a**)

Syntax

```
#include <fstab.h>
```

```
int getfsent_r (FSSent, FSFile, PassNo)
struct fstab * FSSent;
AFILE_t * FSFile;
int * PassNo;

int getfsspec_r (Special, FSSent, FSFile, PassNo)
const char * Special;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int getfsfile_r (File, FSSent, FSFile, PassNo)
const char * File;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;

int getfstype_r (Type, FSSent, FSFile, PassNo)
const char * Type;
struct fstab *FSSent;
AFILE_t *FSFile;
int *PassNo;
```

```
int setfsent_r (FSFile, PassNo)
AFILE_t * FSFile;
int *PassNo;
```

```
int endfsent_r (FSFile)
AFILE_t *FSFile;
```

Description

The **getfsent_r** subroutine reads the next line of the **/etc/filesystems** file, opening it necessary.

The **setfsent_r** subroutine opens the **filesystems** file and positions to the first record.

The **endfsent_r** subroutine closes the **filesystems** file.

The **getfsspec_r** and **getfsfile_r** subroutines search sequentially from the beginning of the file until a matching special file name or file-system file name is found, or until the end of the file is encountered. The **getfstype_r** subroutine behaves similarly, matching on the file-system type field.

Programs using this subroutine must link to the **libpthreads.a** library.

Parameters

Item	Description
FSSent	Points to a structure containing information about the file system. The *FSSent* parameter must be allocated by the caller. It cannot be a null value.
FSFile	Points to an attribute structure. The *FSFile* parameter is used to pass values between subroutines.
PassNo	Points to an integer. The **setfsent_r** subroutine initializes the *PassNo* parameter.
Special	Specifies a special file name to search for in the **filesystems** file.
File	Specifies a file name to search for in the **filesystems** file.
Type	Specifies a type to search for in the **filesystems** file.

Return Values

Item	Description
0	Indicates that the subroutine was successful.
-1	Indicates that the subroutine was not successful.

Files

Item	Description
/etc/filesystems	Centralizes file-system characteristics.

getroles Subroutine

Purpose

Gets the role ID of the current process.

Library

Security Library (**libc.a**)

Syntax

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/cred.h>

int getroles (pid,roles, nroles)
pid_t pid;
rid_t *roles;
int nroles;
```

Description

The **getroles** subroutine gets the supplementary role ID of the process specified by the *pid* parameter. The list is stored in the array pointed to by the *roles* parameter. The *nroles* parameter indicates the number of entries that can be stored in this array. The **getroles** subroutine never returns more than the number of entries specified by the **MAX_ROLES** constant. (The **MAX_ROLES** constant is defined in the <**sys/cred.h**> header file.) If the value in the *nroles* parameter is 0, the **getroles** subroutine returns the number of roles in the given process.

Parameters

Item	Description
Pid	Indicates the process for which the role IDs are requested.
Roles	Points to the array in which the role IDs of the user's process is stored.
nroles	Indicates the number of entries that can be stored in the array pointed to by the *roles* parameter.

Return Values

The **getroles** subroutine returns one of the following values:

Item	Description
0	The subroutine completes successfully.
-1	An error has occurred. An **errno** global variable is set to indicate the error.

Error Codes

The **getroles** subroutine fails if any of the following value is true:

Item	Description
EFAULT	The *roles* and *nroles* parameters specify an array that is partially or completely outside of the process' allocated address space.
EINVAL	The value of the *nroles* parameter is smaller than that of the *roles* parameter in the current process.
EPERM	The invoker does not have the **PV_DAC_RID** privilege in its effective privilege set when the *Pid* is not the same as the current process ID.
ESRCH	No process has a process ID that equals to *Pid*.

getsid Subroutine

Purpose

Returns the session ID of the calling process.

Library

(**libc.a**)

Syntax

#include <unistd.h>

pid_t getsid (pid_ t *pid***)**

Description

The **getsid** subroutine returns the process group ID of the process that is the session leader of the process specified by *pid*. If *pid* is equal to **pid_t** subroutine, it specifies the calling process.

Parameters

Item	Description
pid	A process ID of the process being queried.

Return Values

Upon successful completion, **getsid** subroutine returns the process group ID of the session leaded of the specified process. Otherwise, it returns (**pid_t**)-1 and set **errno** to indicate the error.

Item	Description
id	The session ID of the requested process.
-1	Not successful and the **errno** global variable is set to one of the following error codes.

Error Codes

Item	Description
ESRCH	There is no process with a process ID equal to *pid.*

Item	Description
EPERM	The process specified by pid is not in the same session as the calling process.
ESRCH	There is no process with a process ID equal to *pid*.

getssys Subroutine

Purpose

Reads a subsystem record.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
```

```
int getssys( SubsystemName, SRCSubsystem)
char * SubsystemName;
struct SRCsubsys * SRCSubsystem;
```

Description

The **getssys** subroutine reads a subsystem record associated with the specified subsystem and returns the ODM record in the **SRCsubsys** structure.

The **SRCsubsys** structure is defined in the **sys/srcobj.h** file.

Parameters

Item	Description
SRCSubsystem	Points to the **SRCsubsys** structure.
SubsystemName	Specifies the name of the subsystem to be read.

Return Values

Upon successful completion, the **getssys** subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **odmerrno** variable is set to indicate the error, or an SRC error code is returned.

Error Codes

If the **getssys** subroutine fails, the following is returned:

Item	Description
SRC_NOREC	Subsystem name does not exist.

Files

Item	Description
/etc/objrepos/SRCsubsys	SRC Subsystem Configuration object class.

getsubopt Subroutine

Purpose

Parse suboptions from a string.

Library

Standard C Library **(libc.a)**

Syntax

```
#include <stdlib.h>

int getsubopt (char **optionp,
char * const * tokens,
char ** valuep)
```

Description

The **getsubopt** subroutine parses suboptions in a flag parameter that were initially parsed by the **getopt** subroutine. These suboptions are separated by commas and may consist of either a single token, or a token-value pair separated by an equal sign. Because commas delimit suboptions in the option string, they are not allowed to be part of the suboption or the value of a suboption. similarly, because the equal sign separates a token from its value, a token must not contain an equal sign.

The **getsubopt** subroutine takes the address of a pointer to the option string, a vector of possible tokens, and the address of a value string pointer. It returns the index of the token that matched the suboption in the input string or -1 if there was no match. If the option string at **optionp* contains only one suboption, the **getsubopt** subroutine updates **optionp* to point to the start of the next suboption. It the suboption has an associated value, the **getsubopt** subroutine updates **valuep* to point to the value's first character. Otherwise it sets **valuep* to a NULL pointer.

The token vector is organized as a series of pointers to strings. The end of the token vector is identified by a NULL pointer.

When the **getsubopt** subroutine returns, if **valuep* is not a NULL pointer then the suboption processed included a value. The calling program may use this information to determine if the presence or lack of a value for this suboption is an error.

Additionally, when the **getsubopt** subroutine fails to match the suboption with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

Return Values

The **getsubopt** subroutine returns the index of the matched token string, or -1 if no token strings were matched.

getsubsvr Subroutine

Purpose

Reads a subsystem record.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <sys/srcobj.h>
#include <spc.h>
```

int getsubsvr(*SubserverName*, *SRCSubserver***)**
char ***_SubserverName_;**
struct SRCSubsvr ***_SRCSubserver_;**

Description

The **getsubsvr** subroutine reads a subsystem record associated with the specified subserver and returns the ODM record in the **SRCsubsvr** structure.

The **SRCsubsvr** structure is defined in the **sys/srcobj.h** file and includes the following fields:

Item	Description
char	sub_type[30];
char	subsysname[30];
short	sub_code;

Parameters

Item	Description
SRCSubserver	Points to the **SRCsubsvr** structure.
SubserverName	Specifies the subserver to be read.

Return Values

Upon successful completion, the **getsubsvr** subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **odmerrno** variable is set to indicate the error, or an SRC error code is returned.

Error Codes

If the **getsubsvr** subroutine fails, the following is returned:

Item	Description
SRC_NOREC	The specified **SRCsubsvr** record does not exist.

Files

Item	Description
/etc/objrepos/SRCsubsvr	SRC Subserver Configuration object class.

getsyx Subroutine

Purpose

Retrieves the current coordinates of the virtual screen cursor.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

getsyx(*Y***,** *X***)**
int * *Y***, *** *X***;**

Description

The **getsyx** subroutine retrieves the current coordinates of the virtual screen cursor and stores them in the location specified by Y and X. The current coordinates are those where the cursor was placed after the last call to the **wnoutrefresh**, **pnoutrefresh**, **or wrefresh**, subroutine. If the **leaveok** subroutine was TRUE for the last window refreshed, then the **getsyx** subroutine returns -1 for both X and Y.

If lines have been removed from the top of the screen using the **ripoffline** subroutine, *Y* and *X* include these lines. *Y* and *X* should only be used as arguments for the **setsyx** subroutine.

The **getsyx** subroutine, along with the **setsyx** subroutine, is meant to be used by a user-defined function that manipulates curses windows but wants the position of the cursor to remain the same. Such a function would do the following:

- Call the **getsyx** subroutine to obtain the current virtual cursor coordinates.
- Continue manipulating the windows.
- Call the **wnoutrefresh** subroutine on each window manipulated.
- Reset the current virtual cursor coordinates to the original values with the **setsyx** subroutine.
- Refresh the display with a call to the **doupdate** subroutine.

Parameters

Item	Description
X	Points to the current row position of the virtual screen cursor. A value of -1 indicates the **leaveok** subroutine was TRUE for the last window refreshed.
Y	Points to the current column position of the virtual screen cursor. A value of -1 indicates the **leaveok** subroutine was TRUE for the last window refreshed.

getsystemcfg Subroutine

Purpose

Displays the system configuration information.

Syntax

```
#include <systemcfg.h>
uint64_t getsystemcfg ( int name)
```

Description

Displays the system configuration information.

Parameters

Item	Description
name	Specifies the system variable setting to be returned. Valid values for the *name* parameter are defined in the **systemcfg.h** file.

Return Values

If the value specified by the *name* parameter is system-defined, the **getsystemcfg** subroutine returns the data that is associated with the structure member represented by the *input* parameter. Otherwise, the **getsystemcfg** subroutine will return **UINT64_MAX**, and **errno** will be set.

Error Codes

The **getsystemcfg** subroutine will fail if:

Item	Description
EINVAL	The value of the *name* parameter is invalid.

gettcbattr or puttcbattr Subroutine

Purpose

Accesses the TCB information in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int gettcbattr (Entry, Attribute, Value, Type)
char * Entry;
char * Attribute;
void * Value;
int  Type;
```

```
int puttcbattr (Entry, Attribute, Value, Type)
char *Entry;
char *Attribute;
void *Value;
int Type;
```

Description

These subroutines access Trusted Computing Base (TCB) information.

The **gettcbattr** subroutine reads a specified attribute from the `tcbck` database. If the database is not already open, the subroutine will do an implicit open for reading.

Similarly, the **puttcbattr** subroutine writes a specified attribute into the `tcbck` database. If the database is not already open, the subroutine does an implicit open for reading and writing. Data changed by **puttcbattr** must be explicitly committed by calling the **puttcbattr** subroutine with a *Type* parameter specifying the **SEC_COMMIT** value. Until the data is committed, only **get** subroutine calls within the process will return the written data.

New entries in the `tcbck` databases must first be created by invoking **puttcbattr** with the **SEC_NEW** type.

The `tcbck` database usually defines all the files and programs that are part of the TCB, but the root user or a member of the security group can choose to define only those files considered to be security-relevant.

Parameters

Item	Description
Attribute	Specifies which attribute is read. The following possible values are defined in the **sysck.h** file: **S_ACL** The access control list for the file. Type: **SEC_CHAR**. **S_CHECKSUM** The checksum of the file. Type: **SEC_CHAR**. **S_CLASS** The logical group of the file. The attribute type is **SEC_LIST**. **S_GROUP** The file group. The attribute type is **SEC_CHAR**. **S_LINKS** The hard links to this file. Type: **SEC_LIST**. **S_MODE** The File mode. Type: **SEC_CHAR**. **S_OWNER** The file owner. Type: **SEC_CHAR**. **S_PROGRAM** The associated checking program for the file. Type: **SEC_CHAR**. **S_SIZE** The size of the file in bytes. Type: **SEC_LONG**. **S_SOURCE** The source for the file. Type: **SEC_CHAR**. **S_SYMLINKS** The symbolic links to the file. Type: **SEC_LIST**. **S_TARGET** The target file (if file is a symbolic link). Type: **SEC_CHAR**. **S_TCB** The Trusted Computer Base. The attribute type is **SEC_BOOL**. **S_TYPE** The type of file. The attribute type is **SEC_CHAR**. Additional user-defined attributes may be used and will be stored in the format specified by the *Type* parameter.
Entry	Specifies the name of the file for which an attribute is to be read or written.
Type	Specifies the type of attribute expected. Valid values are defined in the **usersec.h** file and include: **SEC_BOOL** A pointer to an integer (**int ***) that has been cast to a null pointer. **SEC_CHAR** The format of the attribute is a null-terminated character string. **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters. **SEC_LONG** The format of the attribute is a 32-bit integer.

Item	Description
Value	Specifies the address of a pointer for the **gettcbattr** subroutine. The **gettcbattr** subroutine will return the address of a buffer in the pointer. For the **puttcbattr** subroutine, the *Value* parameter specifies the address of a buffer in which the attribute is stored. See the *Type* parameter for more details.

Security

Item	Description
Files Accessed:	

Mode	File
rw	**/etc/security/sysck.cfg** (write access for **puttcbattr**)

Return Values

The **gettcbattr** and **puttcbattr** subroutines, when successfully completed, return a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Note: These subroutines return errors from other subroutines.

These subroutines fail if the following is true:

Item	Description
EACCES	Access permission is denied for the data request.

The **gettcbattr** and **puttcbattr** subroutines fail if one or more of the following are true:

Item	Description
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors may not be detected.
EINVAL	The *Entry* parameter is null or contains a pointer to a null string.
EINVAL	The *Type* parameter contains more than one of the **SEC_BOOL**, **SEC_CHAR**, **SEC_LIST**, or **SEC_LONG** attributes.
EINVAL	The *Type* parameter specifies that an individual attribute is to be committed, and the *Entry* parameter is null.
ENOENT	The specified *Entry* parameter does not exist or the attribute is not defined for this entry.
EPERM	Operation is not permitted.

getthrds Subroutine

Purpose

Gets kernel thread table entries.

Library

Standard C library (**libc.a**)

Syntax

```
#include <procinfo.h>
#include <sys/types.h>
```

```
int
getthrds ( ProcessIdentifier,  ThreadBuffer,  ThreadSize,  IndexPointer,  Count)
pid_t ProcessIdentifier;
struct thrdsinfo *ThreadBuffer;
or struct thrdsinfo64 *ThreadBuffer;
int ThreadSize;
tid_t *IndexPointer;
int Count;
```

```
int
getthrds64 ( ProcessIdentifier,  ThreadBuffer,  ThreadSize,  IndexPointer,  Count)
pid_t ProcessIdentifier;
struct thrdentry64 *ThreadBuffer;
int ThreadSize;
tid64_t *IndexPointer;
int Count;
```

Description

The **getthrds** subroutine returns information about kernel threads, including kernel thread table information defined by the **thrdsinfo** or **thrdsinfo64** structure.

The **getthrds** subroutine retrieves up to *Count* kernel thread table entries, starting with the entry corresponding to the thread identifier indicated by *IndexPointer*, and places them in the array of **thrdsinfo** or **thrdsinfo64**, or **thrdentry64** structures indicated by the *ThreadBuffer* parameter.

On return, the kernel thread identifier referenced by *IndexPointer* is updated to indicate the next kernel thread table entry to be retrieved. The **getthrds** subroutine returns the number of kernel thread table entries retrieved.

If the *ProcessIdentifier* parameter indicates a process identifier, only kernel threads belonging to that process are considered. If this parameter is set to -1, all kernel threads are considered.

The **getthrds** subroutine is normally called repeatedly in a loop, starting with a kernel thread identifier of zero, and looping until the return value is less than *Count*, indicating that there are no more entries to retrieve.

1. Do not use information from the **procsinfo** structure (see the **getprocs** subroutine) to determine the value of the *Count* parameter; a process may create or destroy kernel threads in the interval between a call to **getprocs** and a subsequent call to **getthrds**.
2. The kernel thread table may change while the **getthrds** subroutine is accessing it. Returned entries will always be consistent, but since kernel threads can be created or destroyed while the **getthrds** subroutine is running, there is no guarantee that retrieved entries will still exist, or that all existing kernel threads have been retrieved.

When used in 32-bit mode, limits larger than can be represented in 32 bits are truncated to RLIM_INFINITY. Large values are truncated to INT_MAX. 64-bit applications are required to use **getthrds64()** and **struct thrdentry64.** Note that **struct thrdentry64** contains the same information as **struct thrdsinfo64** with the only difference being support for the 64-bit tid_t and the 256-bit sigset_t. Application developers are also encouraged to use **getthrds64()** in 32-bit applications to obtain 64-bit thread information as this interface provides the new, larger types. The **getthrds()** interface will still be supported for 32-bit applications using **struct thrdsinfo** or **struct thrdsinfo64**, but will not be available to 64-bit applications.

Parameters

ProcessIdentifier
Specifies the process identifier of the process whose kernel threads are to be retrieved. If this parameter is set to -1, all kernel threads in the kernel thread table are retrieved.

ThreadBuffer
Specifies the starting address of an array of **thrdsinfo** or **thrdsinfo64**, or **thrdentry64** structures which will be filled in with kernel thread table entries. If a value of **NULL** is passed for this parameter, the **getthrds** subroutine scans the kernel thread table and sets return values as normal, but no kernel thread table entries are retrieved.

ThreadSize
Specifies the size of a single **thrdsinfo**, **thrdsinfo64**, or **thrdentry64** structure.

IndexPointer
Specifies the address of a kernel thread identifier which indicates the required kernel thread table entry (this does not have to correspond to an existing kernel thread). A kernel thread identifier of zero selects the first entry in the table. The kernel thread identifier is updated to indicate the next entry to be retrieved.

Count
Specifies the number of kernel thread table entries requested.

Return Value

If successful, the **getthrds** subroutine returns the number of kernel thread table entries retrieved; if this is less than the number requested, the end of the kernel thread table has been reached. A value of 0 is returned when the end of the kernel thread table has been reached. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **getthrds** subroutine fails if the following are true:

Item	Description
EINVAL	The *ThreadSize* is invalid, or the *IndexPointer* parameter does not point to a valid kernel thread identifier, or the *Count* parameter is not greater than zero.
ESRCH	The process specified by the *ProcessIdentifier* parameter does not exist.
EFAULT	The copy operation to one of the buffers failed.

gettimeofday, settimeofday, or ftime Subroutine

Purpose

Displays, gets and sets date and time.

Libraries

gettimeofday, **settimeofday**: Standard C Library (l**ibc.a**)

ftime: Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <sys/time.h>
int gettimeofday ( Tp,  Tzp)
struct timeval *Tp;
void *Tzp;
int settimeofday (Tp, Tzp)
```

struct timeval ****Tp*;**
struct timezone ****Tzp*;**

```
#include <sys/types.h>
#include <sys/timeb.h>
int ftime (Tp)
struct timeb *Tp;
```

Description

Current Greenwich time and the current time zone are displayed with the **gettimeofday** subroutine, and set with the **settimeofday** subroutine. The time is expressed in seconds and microseconds since midnight (0 hour), January 1, 1970. The resolution of the system clock is hardware-dependent, and the time may be updated either continuously or in "ticks." If the *Tzp* parameter has a value of 0, the time zone information is not returned or set.

If a recent `adjtime` subroutine call is causing the clock to be adjusted backwards, it is possible that two closely spaced `gettimeofday` calls will observe that time has moved backwards. You can set the `GETTOD_ADJ_MONOTONIC` environment value to cause the returned value to never decrease. After this environment variable is set, the returned value briefly remains constant as necessary to always report a nondecreasing time of day. This extra processing adds significant pathlength to `gettimeofday`. Although any setting of this environment variable requires this extra processing, setting it to 1 is recommended for future compatibility.

The *Tp* parameter returns a pointer to a **timeval** structure that contains the time since the epoch began in seconds and microseconds.

The **timezone** structure indicates both the local time zone (measured in minutes of time westward from Greenwich) and a flag that, if nonzero, indicates that daylight saving time applies locally during the appropriate part of the year.

In addition to the difference in timer granularity, the **timezone** structure distinguishes these subroutines from the POSIX **gettimer** and **settimer** subroutines, which deal strictly with Greenwich Mean Time.

The **ftime** subroutine fills in a structure pointed to by its argument, as defined by **<sys/timeb.h>**. The structure contains the time in seconds since 00:00:00 UTC (Coordinated Universal Time), January 1, 1970, up to 1000 milliseconds of more-precise interval, the local timezone (measured in minutes of time westward from UTC), and a flag that, if nonzero, indicates that Daylight Saving time is in effect, and the values stored in the timeb structure have been adjusted accordingly.

Parameters

Item	Description
Tp	Pointer to a **timeval** structure, defined in the **sys/time.h** file.
Tzp	Pointer to a **timezone** structure, defined in the **sys/time.h** file.

Return Values

If the subroutine succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned and **errno** is set to indicate the error.

Error Codes

If the **settimeofday** subroutine is unsuccessful, the **errno** value is set to **EPERM** to indicate that the process's effective user ID does not have root user authority.

No errors are defined for the **gettimeofday** or **ftime** subroutine.

gettimer, settimer, restimer, stime, or time Subroutine

Purpose

Gets or sets the current value for the specified systemwide timer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>
#include <sys/types.h>
```

```
int gettimer( TimerType,  Value)
timer_t TimerType;
struct timestruc_t * Value;
```

```
#include <sys/timers.h>
#include <sys/types.h>
```

```
int gettimer( TimerType,  Value)
timer_t TimerType;
struct itimerspec * Value;
```

```
int settimer(TimerType,  TimePointer)
int TimerType;
const struct timestruc_t *TimePointer;
```

```
int restimer(TimerType,  Resolution,  MaximumValue)
int TimerType;
struct timestruc_t *Resolution, *MaximumValue;
```

```
int stime( Tp)
long *Tp;
```

```
#include <sys/types.h>
```

```
time_t time(Tp)
time_t *Tp;
```

Description

The **settimer** subroutine is used to set the current value of the *TimePointer* parameter for the systemwide timer, specified by the *TimerType* parameter.

When the **gettimer** subroutine is used with the function prototype in **sys/timers.h**, then except for the parameters, the **gettimer** subroutine is identical to the **getinterval** subroutine. Use of the **getinterval** subroutine is recommended, unless the **gettimer** subroutine is required for a standards-conformant application. The description and semantics of the **gettimer** subroutine are subject to change between releases, pending changes in the draft standard upon which the current **gettimer** subroutine description is based.

When the **gettimer** subroutine is used with the function prototype in **/sys/timers.h**, the **gettimer** subroutine returns an **itimerspec** structure to the pointer specified by the *Value* parameter. The **it_value** member of the **itimerspec** structure represents the amount of time in the current interval before the timer (specified by the *TimerType* parameter) expires, or a zero interval if the timer is disabled. The members of the pointer specified by the *Value* parameter are subject to the resolution of the timer.

When the **gettimer** subroutine is used with the function prototype in **sys/time.h**, the **gettimer** subroutine returns a **timestruc** structure to the pointer specified by the *Value* parameter. This structure holds the current value of the system wide timer specified by the *Value* parameter.

The resolution of any timer can be obtained by the **restimer** subroutine. The *Resolution* parameter represents the resolution of the specified timer. The *MaximumValue* parameter represents the maximum possible timer value. The value of these parameters are the resolution accepted by the **settimer** subroutine.

Note: If a nonprivileged user attempts to submit a fine granularity timer (that is, a timer request of less than 10 milliseconds), the timer request is raised to 10 milliseconds.

The **time** subroutine returns the time in seconds since the Epoch (that is, 00:00:00 GMT, January 1, 1970). The *Tp* parameter points to an area where the return value is also stored. If the *Tp* parameter is a null pointer, no value is stored.

The **stime** subroutine is implemented to provide compatibility with older AIX, AT&T System V, and BSD systems. It calls the **settimer** subroutine using the **TIMEOFDAY** timer.

Parameters

Item	Description
Value	Points to a structure of type **itimerspec**.
TimerType	Specifies the systemwide timer: **TIMEOFDAY** (POSIX system clock timer) This timer represents the time-of-day clock for the system. For this timer, the values returned by the **gettimer** subroutine and specified by the **settimer** subroutine represent the amount of time since 00:00:00 GMT, January 1, 1970.
TimePointer	Points to a structure of type **struct timestruc_t**.
Resolution	The resolution of a specified timer.
MaximumValue	The maximum possible timer value.
Tp	Points to a structure containing the time in seconds.

Return Values

The **gettimer**, **settimer**, **restimer**, and **stime** subroutines return a value of 0 (zero) if the call is successful. A return value of -1 indicates an error occurred, and **errno** is set.

The **time** subroutine returns the value of time in seconds since Epoch. Otherwise, a value of ((**time_t**) - 1) is returned and the **errno** global variable is set to indicate the error.

Error Codes

If an error occurs in the **gettimer**, **settimer**, **restimer**, or **stime** subroutine, a return value of - 1 is received and the **errno** global variable is set to one of the following error codes:

Item	Description
EINVAL	The *TimerType* parameter does not specify a known systemwide timer, or the *TimePointer* parameter of the **settimer** subroutine is outside the range for the specified systemwide timer.
EFAULT	A parameter address referenced memory that was not valid.
EIO	An error occurred while accessing the timer device.

Item	Description
EPERM	The requesting process does not have the appropriate privilege to set the specified timer.

If the **time** subroutine is unsuccessful, a return value of -1 is received and the **errno** global variable is set to the following:

Item	Description
EFAULT	A parameter address referenced memory that was not valid.

gettimerid Subroutine

Purpose

Allocates a per-process interval timer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>
#include <sys/events.h>

timer_t gettimerid( timertype,  notifytype)
int timertype;
int notifytype;
```

Description

The **gettimerid** subroutine is used to allocate a per-process interval timer based on the timer with the given timer type. The unique ID is used to identify the interval timer in interval timer requests. (For more information, see **getinterval** subroutine). The particular timer type, the *timertype* parameter, is defined in the **sys/time.h** file and can identify either a system-wide timer or a per-process timer. The mechanism by which the process is to be notified of the expiration of the timer event is the *notifytype* parameter, which is defined in the **sys/events.h** file.

The *timertype* parameter represents one of the following timer types:

Item	Description
TIMEOFDAY	POSIX system clock timer. This timer represents the time-of-day clock for the system. For this timer, the values returned by the **gettimer** subroutine and specified by the **settimer** subroutine represent the amount of time since 00:00:00 GMT, January 1, 1970, in nanoseconds.
TIMERID_ALRM	Alarm timer. This timer schedules the delivery of a **SIGALRM** signal at a timer specified in the call to the **settimer** subroutine.
TIMERID_REAL	Real-time timer. The real-time timer decrements in real time. A **SIGALRM** signal is delivered when this timer expires.
TIMERID_REAL_TH	Real-time, per-thread timer. Decrements in real time and delivers a **SIGTALRM** signal when it expires. The **SIGTALRM** is sent to the thread that sets the timer. Each thread has its own timer and can manipulate its own timer. This timer is only supported with the 1:1 thread model. If the timer is used in M:N thread model, undefined results might occur.

Item	Description
TIMERID_VIRTUAL	Virtual timer. The virtual timer decrements in process virtual time. it runs only when the process is executing in user mode. A **SIGVTALRM** signal is delivered when it expires.
TIMERID_PROF	Profiling timer. The profiling timer decrements both when running in user mode and when the system is running for the process. It is designed to be used by processes to profile their execution statistically. A **SIGPROF** signal is delivered when the profiling timer expires.

Interval timers with a notification value of **DELIVERY_SIGNAL** are inherited across an **exec** subroutine.

Parameters

Item	Description
notifytype	Notifies the process of the expiration of the timer event.
timertype	Identifies either a system-wide timer or a per-process timer.

Return Values

If the **gettimerid** subroutine succeeds, it returns a **timer_t** structure that can be passed to the per-process interval timer subroutines, such as the **getinterval** subroutine. If an error occurs, the value -1 is returned and **errno** is set.

Error Codes

If the **gettimerid** subroutine fails, the value -1 is returned and **errno** is set to one of the following error codes:

Item	Description
EAGAIN	The calling process has already allocated all of the interval timers associated with the specified timer type for this implementation.
EINVAL	The specified timer type is not defined.

getttyent, getttynam, setttyent, or endttyent Subroutine

Purpose

Gets a tty description file entry.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ttyent.h>

struct ttyent *getttyent()
struct ttyent *getttynam( Name)
char *Name;
void setttyent()
void endttyent()
```

Description

Attention: Do not use the **getttyent**, **getttynam**, **setttyent**, or **endttyent** subroutine in a multithreaded environment.

The **getttyent** and **getttynam** subroutines each return a pointer to an object with the **ttyent** structure. This structure contains the broken-out fields of a line from the tty description file. The **ttyent** structure is in the **/usr/include/sys/ttyent.h** file and contains the following fields:

Item	Description
`tty_name`	The name of the character special file in the **/dev** directory. The character special file must reside in the **/dev** directory.
`ty_getty`	The command that is called by the **init** process to initialize tty line characteristics. This is usually the **getty** command, but any arbitrary command can be used. A typical use is to initiate a terminal emulator in a window system.
`ty_type`	The name of the default terminal type connected to this tty line. This is typically a name from the **termcap** database. The **TERM** environment variable is initialized with this name by the **getty** or **login** command.
`ty_status`	A mask of bit fields that indicate various actions to be allowed on this tty line. The following is a description of each flag: **TTY_ON** Enables logins (that is, the **init** process starts the specified **getty** command on this entry). **TTY_SECURE** Allows a user with root user authority to log in to this terminal. The **TTY_ON** flag must be included.
`ty_window`	The command to execute for a window system associated with the line. The window system is started before the command specified in the `ty_getty` field is executed. If none is specified, this is null.
`ty_comment`	The trailing comment field. A leading delimiter and white space is removed.

The **getttyent** subroutine reads the next line from the tty file, opening the file if necessary. The **setttyent** subroutine rewinds the file. The **endttyent** subroutine closes it.

The **getttynam** subroutine searches from the beginning of the file until a matching name (specified by the *Name* parameter) is found (or until the EOF is encountered).

Parameters

Item	Description
Name	Specifies the name of a tty description file.

Return Values

These subroutines return a null pointer when they encounter an EOF (end-of-file) character or an error.

Files

Item	Description
/usr/lib/libodm.a	Specifies the ODM (Object Data Manager) library.
/usr/lib/libcfg.a	Archives device configuration subroutines.
/etc/termcap	Defines terminal capabilities.

getuid, geteuid, or getuidx Subroutine

Purpose

Gets the real or effective user ID of the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <unistd.h>
```

```
uid_t getuid(void)
```

```
uid_t geteuid(void)
```

```
#include <id.h>
```

```
uid_t getuidx (int type);
```

Description

The **getuid** subroutine returns the real user ID of the current process. The **geteuid** subroutine returns the effective user ID of the current process.

The **getuidx** subroutine returns the user ID indicated by the *type* parameter of the calling process.

These subroutines are part of Base Operating System (BOS) Runtime.

Return Values

The **getuid**, **geteuid** and **getuidx** subroutines return the corresponding user ID. The **getuid** and **geteuid** subroutines always succeed.

The **getuidx** subroutine will return -1 and set the global **errno** variable to EINVAL if the *type* parameter is not one of ID_REAL, ID_EFFECTIVE, ID_SAVED or ID_LOGIN.

Parameters

Item	Description
type	Specifies the user ID to get. Must be one of ID_REAL (real user ID), ID_EFFECTIVE (effective user ID), ID_SAVED (saved set-user ID) or ID_LOGIN (login user ID).

Error Codes

If the **getuidx** subroutine fails the following is returned:

Item	Description
EINVAL	Indicates the value of the type parameter is invalid.

getuinfo Subroutine

Purpose

Finds a value associated with a user.

Library

Standard C Library (**libc.a**)

Syntax

```
char *getuinfo ( Name)
char *Name;
```

Description

The **getuinfo** subroutine finds a value associated with a user. This subroutine searches a user information buffer for a string of the form *Name*=*Value* and returns a pointer to the *Value* substring if the *Name* value is found. A null value is returned if the *Name* value is not found.

The **INuibp** global variable points to the user information buffer:

```
extern char *INuibp;
```

This variable is initialized to a null value.

If the **INuibp** global variable is null when the **getuinfo** subroutine is called, the **usrinfo** subroutine is called to read user information from the kernel into a local buffer. The **INUuibp** is set to the address of the local buffer. If the **INuibp** external variable is not set, the **usrinfo** subroutine is automatically called the first time the **getuinfo** subroutine is called.

Parameter

Item	Description
Name	Specifies a user name.

getuinfox Subroutine

Purpose

Finds a value associated with a user.

Library

Standard C Library (**libc.a**)

Syntax

```
char *getuinfox ( Name)
char *Name;
```

Description

The **getuinfox** subroutine finds a value associated with a user. This subroutine searches a privileged kernel buffer for a string of the form *Name*=*Value* and returns a pointer to the *Value* substring if the *Name*

value is found. A Null value is returned if the *Name* value is not found. The caller is responsible for freeing the memory returned by the **getuinfox** subroutine.

Parameters

Item	Description
Name	Specifies a name.

Return Values

Upon success, the **getuinfox** subroutine returns a pointer to the *Value* substring.

Error Codes

A Null value is returned if the *Name* value is not found.

getuserattr, IDtouser, nextuser, or putuserattr Subroutine

Purpose

Accesses the user information in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getuserattr (User, Attribute, Value, Type)
char * User;
char * Attribute;
void * Value;
int  Type;

char *IDtouser( UID)
uid_t UID;

char *nextuser ( Mode,  Argument)
int Mode, Argument;

int putuserattr (User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description

Attention: These subroutines and the **setpwent** and **setgrent** subroutines should not be used simultaneously. The results can be unpredictable.

These subroutines access user information. Because of their greater granularity and extensibility, you should use them instead of the **getpwent** routines.

The **getuserattr** subroutine reads a specified attribute from the user database. If the database is not already open, this subroutine does an implicit open for reading. A call to the **getuserattr** subroutine for every new user verifies that the user exists.

Similarly, the **putuserattr** subroutine writes a specified attribute into the user database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putuserattr** subroutine must be explicitly committed by calling the **putuserattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only these subroutines within the process return written data.

New entries in the user and group databases must first be created by invoking **putuserattr** with the **SEC_NEW** type.

The **IDtouser** subroutine translates a user ID into a user name.

The **nextuser** subroutine returns the next user in a linear search of the user database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setuserdb** and **enduserdb** subroutines should be used to open and close the user database.

The **enduserdb** subroutine frees all memory allocated by the **getuserattr** subroutine.

Parameters

Argument
Presently unused and must be specified as null.

Attribute
Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file:

S_CORECOMP
Core compression status. The attribute type is **SEC_CHAR**.

S_COREPATH
Core path specification status. The attribute type is **SEC_CHAR**.

S_COREPNAME
Core path specification location. The attribute type is **SEC_CHAR**.

S_CORENAMING
Core naming status. The attribute type is **SEC_CHAR**.

S_ID
User ID. The attribute type is **SEC_INT**.

S_PGID
Principle group ID.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the Lightweight Directory Access Protocol (LDAP) group ID can be assigned to LOCAL user as primary group ID and vice versa.

The attribute type is **SEC_INT**.

S_PGRP
Principle group name.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user as primary group and vice versa.

The attribute type is **SEC_CHAR**.

S_GROUPS
Groups to which the user belongs.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_ADMGROUPS
Groups for which the user is an administrator.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_ADMIN
Administrative status of a user. The attribute type is **SEC_BOOL**.

S_AUDITCLASSES
Audit classes to which the user belongs. The attribute type is **SEC_LIST**.

S_AUTHSYSTEM
Defines the user's authentication method. The attribute type is **SEC_CHAR.**

S_HOME
Home directory. The attribute type is **SEC_CHAR**.

S_SHELL
Initial program run by a user. The attribute type is **SEC_CHAR**.

S_GECOS
Personal information for a user. The attribute type is **SEC_CHAR**.

S_USRENV
User-state environment variables. The attribute type is **SEC_LIST**.

S_SYSENV
Protected-state environment variables. The attribute type is **SEC_LIST**.

S_LOGINCHK
Specifies whether the user account can be used for local logins. The attribute type is **SEC_BOOL**.

S_HISTEXPIRE
Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is **SEC_INT**.

S_HISTSIZE
Specifies the number of previous passwords that the user cannot reuse. The attribute type is **SEC_INT**.

S_MAXREPEAT
Defines the maximum number of times a user can repeat a character in a new password. The attribute type is **SEC_INT**.

S_MINAGE
Defines the minimum age in weeks that the user's password must exist before the user can change it. The attribute type is **SEC_INT**.

S_PWDCHECKS
Defines the password restriction methods for this account. The attribute type is **SEC_LIST**.

S_MINALPHA
Defines the minimum number of alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_MINDIFF
Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is **SEC_INT**.

S_MINLEN
Defines the minimum length of a user's password. The attribute type is **SEC_INT**.

S_MINOTHER
Defines the minimum number of non-alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_DICTION
Defines the password dictionaries for this account. The attribute type is **SEC_LIST**.

S_SUCHK
Specifies whether the user account can be accessed with the **su** command. Type **SEC_BOOL**.

S_REGISTRY
Defines the user's authentication registry. The attribute type is **SEC_CHAR**.

S_RLOGINCHK
Specifies whether the user account can be used for remote logins using the **telnet** or **rlogin** commands. The attribute type is **SEC_BOOL**.

S_DAEMONCHK
Specifies whether the user account can be used for daemon execution of programs and subsystems using the **cron** daemon or **src**. The attribute type is **SEC_BOOL**.

S_TPATH
Defines how the account may be used on the trusted path. The attribute type is **SEC_CHAR**. This attribute must be one of the following values:

nosak
The secure attention key is not enabled for this account.

notsh
The trusted shell cannot be accessed from this account.

always
This account may only run trusted programs.

on
Normal trusted-path processing applies.

S_TTYS
List of ttys that can or cannot be used to access this account. The attribute type is **SEC_LIST**.

S_SUGROUPS
Groups that can or cannot access this account.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_EXPIRATION
Expiration date for this account is a string in the form MMDDhhmmyy, where MM is the month, DD is the day, hh is the hour in 0 to 24 hour notation, mm is the minutes past the hour, and yy is the last two digits of the year. The attribute type is **SEC_CHAR**. For more information about the password expiration, see the **/etc/security/user** file.

S_AUTH1
Primary authentication methods for this account. The attribute type is **SEC_LIST**.

S_AUTH2
Secondary authentication methods for this account. The attribute type is **SEC_LIST**.

S_UFSIZE
Process file size soft limit. The attribute type is **SEC_INT**.

S_UCPU
Process CPU time soft limit. The attribute type is **SEC_INT**.

S_UDATA
Process data segment size soft limit. The attribute type is **SEC_INT**.

S_USTACK
Process stack segment size soft limit. Type: **SEC_INT**.

S_URSS
Process real memory size soft limit. Type: **SEC_INT**.

S_UCORE
Process core file size soft limit. The attribute type is **SEC_INT**.

S_UNOFILE
Process file descriptor table size soft limit. The attribute type is **SEC_INT**.

S_PWD
Specifies the value of the passwd field in the **/etc/passwd** file. The attribute type is **SEC_CHAR**.

S_UMASK
File creation mask for a user. The attribute type is **SEC_INT**.

S_LOCKED
Specifies whether the user's account can be logged into. The attribute type is **SEC_BOOL**.

S_ROLES
Defines the administrative roles for this account. The attribute type is **SEC_LIST**.

S_UFSIZE_HARD
Process file size hard limit. The attribute type is **SEC_INT**.

S_UCPU_HARD
Process CPU time hard limit. The attribute type is **SEC_INT**.

S_UDATA_HARD
Process data segment size hard limit. The attribute type is **SEC_INT**.

S_USREXPORT
Specifies if the DCE registry can overwrite the local user information with the DCE user information during a DCE export operation. The attribute type is **SEC_BOOL**.

S_USTACK_HARD
Process stack segment size hard limit. Type: **SEC_INT**.

S_URSS_HARD
Process real memory size hard limit. Type: **SEC_INT**.

S_UCORE_HARD
Process core file size hard limit. The attribute type is **SEC_INT**.

S_UNOFILE_HARD
Process file descriptor table size hard limit. The attribute type is **SEC_INT**.

S_DOMAINS
The domains for the user. It can be one or more. The attribute type is **SEC_LIST**.

S_DFLT_ROLES
The default roles for the user. It can be one or more roles. The attribute type is **SEC_LIST**.

S_MINLOWERALPHA
Defines the minimum number of lowercase alphabetic characters required in a new user password. The attribute type is **SEC_INT**.

S_MINUPPERALPHA
Defines the minimum number of uppercase alphabetic characters required in a new user password. The attribute type is **SEC_INT**.

S_MINDIGIT
Defines the minimum number of digits required in a new user password. The attribute type is **SEC_INT**.

S_MINSPECIALCHAR
Defines the minimum number of special characters required in a new user's password. The attribute type is **SEC_INT**.

Note: These values are string constants that should be used by applications both for convenience and to permit optimization in latter implementations. Additional user-defined attributes may be used and will be stored in the format specified by the *Type* parameter.

Mode
Specifies the search mode. This parameter can be used to delimit the search to one or more user credentials databases. Specifying a non-null *Mode* value also implicitly rewinds the search. A null *Mode* value continues the search sequentially through the database. This parameter must include one of the following values specified as a bit mask; these are defined in the **usersec.h** file:

S_LOCAL
Locally defined users are included in the search.

S_SYSTEM
All credentials servers for the system are searched.

Type
Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include:

SEC_INT
The format of the attribute is an integer.

For the **getuserattr** subroutine, the user should supply a pointer to a defined integer variable. For the **putuserattr** subroutine, the user should supply an integer.

SEC_CHAR
The format of the attribute is a null-terminated character string.

For the **getuserattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putuserattr** subroutine, the user should supply a character pointer.

SEC_LIST
The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters.

For the **getuserattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putuserattr** subroutine, the user should supply a character pointer.

SEC_BOOL
The format of the attribute from **getuserattr** is an integer with the value of either 0 (false) or 1 (true). The format of the attribute for **putuserattr** is a null-terminated string containing one of the following strings: true, false, yes, no, always, or never.

For the **getuserattr** subroutine, the user should supply a pointer to a defined integer variable. For the **putuserattr** subroutine, the user should supply a character pointer.

SEC_COMMIT
For the **putuserattr** subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage.

SEC_DELETE
The corresponding attribute is deleted from the database.

SEC_NEW
Updates all the user database files with the new user name when using the **putuserattr** subroutine.

UID
Specifies the user ID to be translated into a user name.

User
Specifies the name of the user for which an attribute is to be read.

Value
Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Security

Item	Description
Files Accessed:	

Mode	File
rw	**/etc/passwd**
rw	**/etc/group**
rw	**/etc/security/user**
rw	**/etc/security/limits**
rw	**/etc/security/group**
rw	**/etc/security/environ**

Return Values

If successful, the **getuserattr** subroutine and the **putuserattr** subroutine return 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

If successful, the **IDtouser** and the **nextuser** subroutines return a character pointer to a buffer containing the requested user name. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

If any of these subroutines fail, the following is returned:

Item	Description
EACCES	Access permission is denied for the data request.

If the **getuserattr** subroutine or the **getuserattrs** subroutine fail, the following is returned:

Item	Description
EIO	Failed to access remote user database.

If the **getuserattr** and **putuserattr** subroutines fail, one or more of the following is returned:

Item	Description
ENOENT	The specified *User* parameter does not exist.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or null.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors may not be detected.
EPERM	Operation is not permitted.
ENOATTR	The specified attribute is not defined for this user.

If the **IDtouser** subroutine fails, one or more of the following is returned:

Item	Description
ENOENT	The specified *User* parameter does not exist

If the **nextuser** subroutine fails, one or more of the following is returned:

Item	Description
EINVAL	The *Mode* parameter is not one of null, **S_LOCAL**, or **S_SYSTEM**.
EINVAL	The *Argument* parameter is not null.
ENOENT	The end of the search was reached.

Files

Item	Description
/etc/passwd	Contains user IDs.

getuserattrs Subroutine

Purpose

Retrieves multiple user attributes in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int getuserattrs (User, Attributes, Count)
char * User;
dbattr_t * Attributes;
int Count
```

Description

Attention: Do not use this subroutine and the **setpwent** and **setgrent** subroutines simultaneously. The results can be unpredictable.

The **getuserattrs** subroutine accesses user information. Because of its greater granularity and extensibility, use it instead of the **getpwent** routines.

The **getuserattrs** subroutine reads one or more attributes from the user database. If the database is not already open, this subroutine does an implicit open for reading. A call to the **getuserattrs** subroutine with an *Attributes* parameter of null and the *Count* parameter of zero for every new user verifies that the user exists.

The *Attributes* array contains information about each attribute that is to be read. The **dbattr_t** data structure contains the following fields:

attr_name
The name of the desired attribute.

attr_idx
Used internally by the **getuserattrs** subroutine.

attr_type
The type of the desired attribute. The following possible attributes are defined in the **usersec.h** file:

S_CORECOMP
Core compression status. The attribute type is **SEC_CHAR**.

S_COREPATH
Core path specification status. The attribute type is **SEC_CHAR**.

S_COREPNAME
Core path specification location. The attribute type is **SEC_CHAR**.

S_CORENAMING
Core naming status. The attribute type is **SEC_CHAR**.

S_ID
User ID. The attribute type is **SEC_INT**.

S_PGID
Principle group ID.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the Lightweight Directory Access Protocol (LDAP) group ID can be assigned to LOCAL user as primary group ID and vice versa.

The attribute type is **SEC_INT**.

S_PGRP
Principle group name.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user as primary group and vice versa.

The attribute type is **SEC_CHAR**.

S_GROUPS
Groups to which the user belongs.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_ADMGROUPS
Groups for which the user is an administrator.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_ADMIN
Administrative status of a user. The attribute type is **SEC_BOOL**.

S_AUDITCLASSES
Audit classes to which the user belongs. The attribute type is **SEC_LIST**.

S_AUTHSYSTEM
Defines the user's authentication method. The attribute type is **SEC_CHAR.**

S_HOME
Home directory. The attribute type is **SEC_CHAR**.

S_SHELL
Initial program run by a user. The attribute type is **SEC_CHAR**.

S_GECOS
Personal information for a user. The attribute type is **SEC_CHAR**.

S_USRENV
User-state environment variables. The attribute type is **SEC_LIST**.

S_SYSENV
Protected-state environment variables. The attribute type is **SEC_LIST**.

S_LOGINCHK
Specifies whether the user account can be used for local logins. The attribute type is **SEC_BOOL**.

S_HISTEXPIRE
Defines the period of time (in weeks) that a user cannot reuse a password. The attribute type is **SEC_INT**.

S_HISTSIZE
Specifies the number of previous passwords that the user cannot reuse. The attribute type is **SEC_INT**.

S_MAXREPEAT
Defines the maximum number of times a user can repeat a character in a new password. The attribute type is **SEC_INT**.

S_MINAGE
Defines the minimum age in weeks that the user's password must exist before the user can change it. The attribute type is **SEC_INT**.

S_PWDCHECKS
Defines the password restriction methods for this account. The attribute type is **SEC_LIST**.

S_MINALPHA
Defines the minimum number of alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_MINDIFF
Defines the minimum number of characters required in a new password that were not in the old password. The attribute type is **SEC_INT**.

S_MINLEN
Defines the minimum length of a user's password. The attribute type is **SEC_INT**.

S_MINOTHER
Defines the minimum number of non-alphabetic characters required in a new user's password. The attribute type is **SEC_INT**.

S_DICTIONLIST
Defines the password dictionaries for this account. The attribute type is **SEC_LIST**.

S_SUCHK
Specifies whether the user account can be accessed with the **su** command. Type **SEC_BOOL**.

S_REGISTRY
Defines the user's authentication registry. The attribute type is **SEC_CHAR**.

S_RLOGINCHK
Specifies whether the user account can be used for remote logins using the **telnet** or **rlogin** commands. The attribute type is **SEC_BOOL**.

S_DAEMONCHK
Specifies whether the user account can be used for daemon execution of programs and subsystems using the **cron** daemon or **src**. The attribute type is **SEC_BOOL**.

S_TPATH
Defines how the account might be used on the trusted path. The attribute type is **SEC_CHAR**. This attribute must be one of the following values:

nosak
The secure attention key is not enabled for this account.

notsh
The trusted shell cannot be accessed from this account.

always
This account may only run trusted programs.

on
Normal trusted-path processing applies.

S_TTYS
List of ttys that can or cannot be used to access this account. The attribute type is **SEC_LIST**.

S_SUGROUPS
Groups that can or cannot access this account.

If the *domainlessgroups* attribute is set in the **/etc/secvars.cfg** file, the LDAP group can be assigned to LOCAL user and vice versa.

The attribute type is **SEC_LIST**.

S_EXPIRATION
Expiration date for this account is a string in the form MMDDhhmmyy, where MM is the month, DD is the day, hh is the hour in 0 to 24 hour notation, mm is the minutes past the hour, and yy is the last two digits of the year. The attribute type is **SEC_CHAR**.

S_AUTH1
Primary authentication methods for this account. The attribute type is **SEC_LIST**.

S_AUTH2
Secondary authentication methods for this account. The attribute type is **SEC_LIST**.

S_UFSIZE
Process file size soft limit. The attribute type is **SEC_INT**.

S_UCPU
Process processor time soft limit. The attribute type is **SEC_INT**.

S_UDATA
Process data segment size soft limit. The attribute type is **SEC_INT**.

S_USTACK
Process stack segment size soft limit. Type: **SEC_INT**.

S_URSS
Process real memory size soft limit. Type: **SEC_INT**.

S_UCORE
Process core file size soft limit. The attribute type is **SEC_INT**.

S_UNOFILE
Process file descriptor table size soft limit. The attribute type is **SEC_INT**.

S_PWD
Specifies the value of the passwd field in the **/etc/passwd** file. The attribute type is **SEC_CHAR**.

S_UMASK
File creation mask for a user. The attribute type is **SEC_INT**.

S_LOCKED
Specifies whether the user's account can be logged into. The attribute type is **SEC_BOOL**.

S_ROLES
Defines the administrative roles for this account. The attribute type is **SEC_LIST**.

S_UFSIZE_HARD
Process file size hard limit. The attribute type is **SEC_INT**.

S_UCPU_HARD
Process processor time hard limit. The attribute type is **SEC_INT**.

S_UDATA_HARD
Process data segment size hard limit. The attribute type is **SEC_INT**.

S_USREXPORT
Specifies if the DCE registry can overwrite the local user information with the DCE user information during a DCE export operation. The attribute type is **SEC_BOOL**.

S_USTACK_HARD
Process stack segment size hard limit. Type: **SEC_INT**.

S_URSS_HARD
Process real memory size hard limit. Type: **SEC_INT**.

S_UCORE_HARD
Process core file size hard limit. The attribute type is **SEC_INT**.

S_UNOFILE_HARD
Process file descriptor table size hard limit. The attribute type is **SEC_INT**.

S_DFLT_ROLES
The default roles for the user. It can be one or more. The attribute type is **SEC_LIST**.

S_DOMAINS
The domains for the user. It can be one or more. The attribute type is **SEC_LIST**.

S_MINLOWERALPHA
Defines the minimum number of lowercase alphabetic characters required in a new user password. The attribute type is **SEC_INT**.

S_MINUPPERALPHA
Defines the minimum number of uppercase alphabetic characters required in a new user password. The attribute type is **SEC_INT**.

S_MINDIGIT
Defines the minimum number of digits required in a new user password. The attribute type is **SEC_INT**.

S_MINSPECIALCHAR
Defines the minimum number of special characters required in a new user password. The attribute type is **SEC_INT**.

attr_flag
The results of the request to read the desired attribute.

attr_un
A union containing the returned values. Its union members, which follows, correspond to the definitions of the **attr_char**, **attr_int**, **attr_long**, and **attr_llong** macros, respectively:

au_char
Attributes of type **SEC_CHAR** and **SEC_LIST** store a pointer to the returned attribute in this member when the requested attribute is successfully read. The caller is responsible for freeing this memory.

au_int
Attributes of type **SEC_INT** and **SEC_BOOL** store the value of the attribute in this member when the requested attribute is successfully read.

au_long
Attributes of type **SEC_LONG** store the value of the attribute in this member when the requested attribute is successfully read.

au_llong
Attributes of type **SEC_LLONG** store the value of the attribute in this member when the requested attribute is successfully read.

attr_domain
The authentication domain containing the attribute. The **getuserattrs** subroutine is responsible for managing the memory referenced by this pointer. If **attr_domain** is specified for an attribute, the get request is sent only to that domain. If **attr_domain** is not specified (that is, set to NULL), **getuserattrs** searches the domains known to the system and sets this field to the name of the domain from which the value is retrieved. This search space can be restricted with the **setauthdb** subroutine so that only the domain specified in the **setauthdb** call is searched. If the request for a NULL domain was not satisfied, the request is tried from the local files using the default stanza.

Use the **setuserdb** and **enduserdb** subroutines to open and close the user database. Failure to explicitly open and close the user database can result in loss of memory and performance.

Parameters

Item	Description
User	Specifies the name of the user for which the attributes are to be read.
Attributes	A pointer to an array of zero or more elements of type **dbattr_t**. The list of user attributes is defined in the **usersec.h** header file.
Count	The number of array elements in *Attributes*.

Security

Files accessed:

Item	Description
Mode	**File**
rw	**/etc/passwd**
rw	**/etc/group**
rw	**/etc/security/user**
rw	**/etc/security/limits**
rw	**/etc/security/group**
rw	**/etc/security/environ**

Return Values

If *User* exists, the **getuserattrs** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error. Each element in the *Attributes* array must be examined on a successful call to **getuserattrs** to determine if the *Attributes* array entry was successfully retrieved.

Error Codes

The **getuserattrs** subroutine returns an error that indicates that the user does or does not exist. Additional errors can indicate an error querying the information databases for the requested attributes.

Item	Description
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Attributes* parameter is null and the *Count* parameter is greater than zero.
ENOENT	The specified *User* parameter does not exist.
EIO	Failed to access remote user database.

If the **getuserattrs** subroutine fails to query an attribute, one or more of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The user does not have access to the attribute specified in the *attr_name* field.
EINVAL	The **attr_type** field in the *Attributes* entry contains a type that is not valid.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
ENOATTR	The **attr_name** field in the *Attributes* entry specifies an attribute that is not defined for this user or group.

Examples

The following sample test program displays the output to a call to **getuserattrs**. In this example, the system has a user named `foo`.

```
#include <stdio.h>
#include <usersec.h>

#define NATTR 3
#define USERNAME "foo"

main() {

    dbattr_t attributes[NATTR];
```

```
    int     i;
    int     rc;


    memset (&attributes, 0, sizeof(attributes));

    /*
     * Fill in the attributes array with "id", "expires" and
     * "SYSTEM" attributes.
     */

    attributes[0].attr_name = S_ID;
    attributes[0].attr_type = SEC_INT;;

    attributes[1].attr_name = S_ADMIN;
    attributes[1].attr_type = SEC_BOOL;

    attributes[2].attr_name = S_AUTHSYSTEM;
    attributes[2].attr_type = SEC_CHAR;

    /*
     * Make a call to getuserattrs.
     */

        setuserdb(S_READ);

    rc = getuserattrs(USERNAME, attributes, NATTR);

        enduserdb();

    if (rc) {
        printf("getuserattrs failed ....\n");
        exit(-1);
    }

    for (i = 0; i < NATTR; i++) {
        printf("attribute name   : %s \n", attributes[i].attr_name);
        printf("attribute flag   : %d \n", attributes[i].attr_flag);

        if (attributes[i].attr_flag) {

            /*
             * No attribute value. Continue.
             */
            printf("\n");
            continue;
        }
        /*
         * We have a value.
         */
        printf("attribute domain : %s \n", attributes[i].attr_domain);
        printf("attribute value  : ");

        switch (attributes[i].attr_type)
        {
            case SEC_CHAR:
                if (attributes[i].attr_char) {
                                    printf("%s\n", attributes[i].attr_char);
                    free(attributes[i].attr_char);
                }
                                break;
            case SEC_INT:
            case SEC_BOOL:
                                printf("%d\n", attributes[i].attr_int);
                                break;
            default:
                break;
                }
        printf("\n");
    }
    exit(0);
}
```

The following output for the call is expected:

```
    attribute name   : id
    attribute flag   : 0
    attribute domain : files
    attribute value  : 206
```

```
    attribute name   : admin
    attribute flag   : 0
    attribute domain : files
    attribute value  : 0

    attribute name   : SYSTEM
    attribute flag   : 0
    attribute domain : files
    attribute value  : compat
```

Files

Item	Description
/etc/passwd	Contains user IDs.

GetUserAuths Subroutine

Purpose

Accesses the set of authorizations of a user.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
char *GetUserAuths(void);
```

Description

The **GetUserAuths** subroutine returns the list of authorizations associated with the real user ID and group set of the process. By default, the ALL authorization is returned for the root user.

Return Values

If successful, the **GetUserAuths** subroutine returns a list of authorizations associated with the user. The format of the list is a series of concatenated strings, each null-terminated. A null string terminates the list. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error.

getuserpw, putuserpw, or putuserpwhist Subroutine

Purpose

Accesses the user authentication data.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpw.h>
```

```
struct userpw *getuserpw ( User)
char *User;

int putuserpw ( Password)
struct userpw *Password;

int putuserpwhist ( Password,  Message)
struct userpw *Password;
char **Message;
```

Description

These subroutines may be used to access user authentication information. Because of their greater granularity and extensibility, you should use them instead of the **getpwent** routines.

The **getuserpw** subroutine reads the user's locally defined password information. If the **setpwdb** subroutine has not been called, the **getuserpw** subroutine will call it as `setpwdb (S_READ)`. This can cause problems if the **putuserpw** subroutine is called later in the program.

The **putuserpw** subroutine updates or creates a locally defined password information stanza in the **/etc/security/passwd** file. The password entry created by the **putuserpw** subroutine is used only if there is an `!` (exclamation point) in the **/etc/passwd** file's `password` field. The user application can use the **putuserattr** subroutine to add an ! to this field.

The **putuserpw** subroutine will open the authentication database read/write if no other access has taken place, but the program should call `setpwdb (S_READ | S_WRITE)` before calling the **putuserpw** subroutine.

The **putuserpwhist** subroutine updates or creates a locally defined password information stanza in the **etc/security/passwd** file. The subroutine also manages a database of previous passwords used for password reuse restriction checking. It is recommended to use the **putuserpwhist** subroutine, rather than the **putuserpw** subroutine, to ensure the password is added to the password history database.

Parameters

Item	Description
Password	Specifies the password structure used to update the password information for this user. This structure is defined in the **userpw.h** file and contains the following members: **upw_name** Specifies the user's name. (The first eight characters must be unique, since longer names are truncated.) **upw_passwd** Specifies the user's password. **upw_lastupdate** Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, January 1, 1970), when the password was last updated. **upw_flags** Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the **userpw.h** file. **PW_NOCHECK** Specifies that new passwords need not meet password restrictions in effect for the system. **PW_ADMCHG** Specifies that the password was last set by an administrator and must be changed at the next successful use of the **login** or **su** command**.** **PW_ADMIN** Specifies that password information for this user may only be changed by the root user.
Message	Indicates a message that specifies an error occurred while updating the password history database. Upon return, the value is either a pointer to a valid string within the memory allocated storage or a null pointer.
User	Specifies the name of the user for which password information is read. (The first eight characters must be unique, since longer names are truncated.)

Security

Files Accessed:

Mode	File
rw	**/etc/security/passwd**

Return Values

If successful, the **getuserpw** subroutine returns a valid pointer to a **userpw** structure. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error. If the user exists but there is no user entry in the **/etc/security/passwd** file, the **getuserpw** subroutine returns success with the name field set to user name, the password field set to NULL, the lastupdate field set to 0 and the flags field set to 0. If the user exists and there is an entry in the **/etc/security/passwd** file but one or more fields are missing, the **getuserpw** subroutine returns the fields that exist.

If the user exists but there is no user entry in the **/etc/security/passwd** file, the **putuserpw** subroutine creates a user stanza in the **/etc/security/passwd** file. If the user exists and there is an entry in the **/etc/security/passwd** file but one or more fields are missing, the **putuserpw** subroutine updates the fields that exist and creates the fields that are missing.

If successful, the **putuserpwhist** subroutine returns a value of 0. If the subroutine failed to update or create a locally defined password information stanza in the **/etc/security/passwd** file, the

putuserpwhist subroutine returns a nonzero value. If the subroutine was unable to update the password history database, a message is returned in the *Message* parameter and a return code of 0 is returned. If the user exists but there is no user entry in the **/etc/security/passwd** file, the **putuserpwhist** subroutine creates a user stanza in the **/etc/security/passwd** file and updates the password history. If the user exists and there is an entry in the **/etc/security/passwd** file but one or more fields are missing, the **putuserpwhist** subroutine updates the fields that exist, creates the fields that are missing and modifies the password history.

Error Codes

The **getuserpw, putuserpw,** and **putuserpwhist** subroutines fail if the following values are true:

Item	Description
EACCES	The user is not able to open the files that contain the password attributes.
ENOENT	The user does not exist in the **/etc/passwd** file.

Subroutines invoked by the **getuserpw, putuserpw,** or **putuserpwhist** subroutines can also set errors.

Files

Item	Description
/etc/security/passwd	Contains user passwords.

getuserpwx Subroutine

Purpose

Accesses the user authentication data.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpw.h>

struct userpwx *getuserpwx (User)
char * User;
```

Description

The **getuserpwx** subroutine accesses user authentication information. Because of its greater granularity and extensibility, use it instead of the **getpwent** routines.

The **getuserpwx** subroutine reads the user's password information from the local administrative domain or from a loadable authentication module that supports the required user attributes.

The **getuserpw** subroutine opens the authentication database read-only if no other access has taken place, but the program should call **setpwdb** (**S_READ**) followed by **endpwdb** after access to the authentication database is no longer required.

The data returned by **getuserpwx** is stored in allocated memory and must be freed by the caller when the data is no longer required. The entire structure can be freed by invoking the **free** subroutine with the pointer returned by **getuserpwx**.

Parameters

Item	Description
User	Specifies the name of the user for which password information is read.

Security

Files accessed:

Item	Description
Mode	**File**
r	**/etc/passwd**
r	**/etc/security/passwd**

Return Values

If successful, the **getuserpwx** subroutine returns a valid pointer to a **userpwx** structure. Otherwise, a null pointer is returned and the **errno** global variable is set to indicate the error. The fields in a **userpwx** structure are defined in the **userpw.h** file, and they include the following members:

Item	Description
upw_name	Specifies the user's name.
upw_passwd	Specifies the user's encrypted password.
upw_lastupdate	Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, 1 January 1970), when the password was last updated.
upw_flags	Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the **userpw.h** file: **PW_NOCHECK** Specifies that new passwords need not meet password restrictions in effect for the system. **PW_ADMCHG** Specifies that the password was last set by an administrator and must be changed at the next successful use of the login or **su** command. **PW_ADMIN** Specifies that password information for this user can only be changed by the root user.
upw_authdb	Specifies the administrative domain containing the authentication data.

Error Codes

The **getuserpwx** subroutine fails if one of the following values is true:

Item	Description
EACCES	The user is not able to open the files that contain the password attributes.
ENOENT	The user does not have an entry in the **/etc/security/passwd** file or other administrative domain.

Subroutines invoked by the **getuserpwx** subroutine can also set errors.

Files

Item	Description
/etc/security/passwd	Contains user passwords.

getusraclattr, nextusracl or putusraclattr Subroutine

Purpose

Accesses the user screen information in the SMIT ACL database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int getusraclattr(User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

```
char *nextusracl(void)
```

```
int putusraclattr(User, Attribute, Value, Type)
char *User;
char *Attribute;
void *Value;
int Type;
```

Description

The **getusraclattr** subroutine reads a specified user attribute from the SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading.

Similarly, the **putusraclattr** subroutine writes a specified attribute into the user SMIT ACL database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by the **putusraclattr** subroutine must be explicitly committed by calling the **putusraclattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getusraclattr** subroutine within the process returns written data.

The **nextusracl** subroutine returns the next user in a linear search of the user SMIT ACL database. The consistency of consecutive searches depends upon the underlying storage-access mechanism and is not guaranteed by this subroutine.

The **setacldb** and **endacldb** subroutines should be used to open and close the database.

Parameters

Item	Description
Attribute	Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file: **S_SCREENS** String of SMIT screens. The attribute type is **SEC_LIST**. **S_ACLMODE** String specifying the SMIT ACL database search scope. The attribute type is **SEC_CHAR**. **S_FUNCMODE** String specifying the databases to be searched. The attribute type is **SEC_CHAR**.
Type	Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include: **SEC_CHAR** The format of the attribute is a null-terminated character string. For the **getusraclattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putusraclattr** subroutine, the user should supply a character pointer. **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series must be an empty (zero character count) string. For the **getusraclattr** subroutine, the user should supply a pointer to a defined character pointer variable. For the **putusraclattr** subroutine, the user should supply a character pointer. **SEC_COMMIT** For the **putusraclattr** subroutine, this value specified by itself indicates that changes to the named user are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no user is specified, the changes to all modified users are committed to permanent storage. **SEC_DELETE** The corresponding attribute is deleted from the user SMIT ACL database. **SEC_NEW** Updates the user SMIT ACL database file with the new user name when using the **putusraclattr** subroutine.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Return Values

If successful, the **getusraclattr** returns 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Possible return codes are:

Item	Description
EACCES	Access permission is denied for the data request.
ENOENT	The specified User parameter does not exist or the attribute is not defined for this user.

Item	Description
ENOATTR	The specified user attribute does not exist for this user.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or null.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
EPERM	Operation is not permitted.

getutent, getutid, getutline, pututline, setutent, endutent, or utmpname Subroutine

Purpose

Accesses **utmp** file entries.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid ( ID)
struct utmp *ID;

struct utmp *getutline ( Line)
struct utmp *Line;

void pututline ( Utmp)
struct utmp *Utmp;

void setutent ( )

void endutent ( )

void utmpname ( File)
char *File;
```

Description

The **getutent**, **getutid**, and **getutline** subroutines return a pointer to a structure of the following type:

```
struct utmp
> {
>     char ut_user[256];           /* User name */
>     char ut_id[14];              /* /etc/inittab ID */
>     char ut_line[64];            /* Device name (console, lnxx) */
>     pid_t ut_pid;                /* Process ID */
>     short ut_type;               /* Type of entry */
>         int __time_t_space;      /* for 32vs64-bit time_t PPC */
>         time_t ut_time;          /* time entry was made */
>         struct exit_status
>         {
>             short e_termination; /* Process termination status */
>             short e_exit;        /* Process exit status */
>         }
```

```
>         ut_exit;                /* The exit status of a process
>                                 /* marked as DEAD_PROCESS. */
>         char ut_host[256];      /* host name */
>         int __dbl_word_pad;     /* for double word alignment */
>         int __reservedA[2];
>         int __reservedV[6];
> };
```

The **getutent** subroutine reads the next entry from a **utmp**-like file. If the file is not open, this subroutine opens it. If the end of the file is reached, the **getutent** subroutine fails.

The **pututline** subroutine writes the supplied *Utmp* parameter structure into the **utmp** file. It is assumed that the user of the **pututline** subroutine has searched for the proper entry point using one of the **getut** subroutines. If not, the **pututline** subroutine calls **getutid** to search forward for the proper place. If so, **pututline** does not search. If the **pututline** subroutine does not find a matching slot for the entry, it adds a new entry to the end of the file.

The **setutent** subroutine resets the input stream to the beginning of the file. Issue a **setuid** call before each search for a new entry if you want to examine the entire file.

The **endutent** subroutine closes the file currently open.

The **utmpname** subroutine changes the name of a file to be examined from **/etc/utmp** to any other file. The name specified is usually **/var/adm/wtmp**. If the specified file does not exist, no indication is given. You are not aware of this fact until your first attempt to reference the file. The **utmpname** subroutine does not open the file. It closes the old file, if currently open, and saves the new file name.

The most current entry is saved in a static structure. To make multiple accesses, you must copy or use the structure between each access. The **getutid** and **getutline** subroutines examine the static structure first. If the contents of the static structure match what they are searching for, they do not read the **utmp** file. Therefore, you must fill the static structure with zeros after each use if you want to use these subroutines to search for multiple occurrences.

If the **pututline** subroutine finds that it is not already at the correct place in the file, the implicit read it performs does not overwrite the contents of the static structure returned by the **getutent** subroutine, the **getuid** subroutine, or the **getutline** subroutine. This allows you to get an entry with one of these subroutines, modify the structure, and pass the pointer back to the **pututline** subroutine for writing.

These subroutines use buffered standard I/O for input. However, the **pututline** subroutine uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the **utmp** and **wtmp** files.

Parameters

Item	Description
ID	If you specify a type of **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, or **NEW_TIME** in the *ID* parameter, the **getutid** subroutine searches forward from the current point in the **utmp** file until an entry with a `ut_type` matching `ID->ut_type` is found. If you specify a type of **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS** in the *ID* parameter, the **getutid** subroutine returns a pointer to the first entry whose type is one of these four and whose `ut_id` field matches `Id->ut_id`. If the end of the file is reached without a match, the **getutid** subroutine fails.
Line	The **getutline** subroutine searches forward from the current point in the **utmp** file until it finds an entry of type **LOGIN_PROCESS** or **USER_PROCESS** that also has a `ut_line` string matching the `Line->ut_line` parameter string. If the end of file is reached without a match, the **getutline** subroutine fails.
Utmp	Points to the **utmp** structure.
File	Specifies the name of the file to be examined.

Return Values

These subroutines fail and return a null pointer if a read or write fails due to a permission conflict or because the end of the file is reached.

Files

Item	Description
/etc/utmp	Path to the **utmp** file, which contains a record of users logged into the system.
/var/adm/wtmp	Path to the **wtmp** file, which contains accounting information about users logged in.

getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag, setvfsent, or endvfsent Subroutine

Purpose

Gets a **vfs** file entry.

Library

Standard C Library(**libc.a**)

Syntax

```
#include <sys/vfs.h>
#include <sys/vmount.h>
```

```
struct vfs_ent *getvfsent( )
```

struct vfs_ent *getvfsbytype(*vfsType***)**
int *vfsType***;**

struct vfs_ent *getvfsbyname(*vfsName***)**
char **vfsName***;**

struct vfs_ent *getvfsbyflag(*vfsFlag***)**
int *vfsFlag***;**

```
void setvfsent( )
```

```
void endvfsent( )
```

Description

Attention: All information is contained in a static area and so must be copied to be saved.

The **getvfsent** subroutine, when first called, returns a pointer to the first **vfs_ent** structure in the file. On the next call, it returns a pointer to the next **vfs_ent** structure in the file. Successive calls are used to search the entire file.

The **vfs_ent** structure is defined in the **vfs.h** file and it contains the following fields:

```
char vfsent_name;
int vfsent_type;
```

```
int vfsent_flags;
char *vfsent_mnt_hlpr;
char *vfsent_fs_hlpr;
```

The **getvfsbytype** subroutine searches from the beginning of the file until it finds a **vfs** type matching the *vfsType* parameter. The subroutine then returns a pointer to the structure in which it was found.

The **getvfsbyname** subroutine searches from the beginning of the file until it finds a **vfs** name matching the *vfsName* parameter. The search is made using flattened names; the search-string uses ASCII equivalent characters.

The **getvfsbytype** subroutine searches from the beginning of the file until it finds a type matching the *vfsType* parameter.

The **getvfsbyflag** subroutine searches from the beginning of the file until it finds the entry whose flag corresponds flags defined in the **vfs.h** file. Currently, these are **VFS_DFLT_LOCAL** and **VFS_DFLT_REMOTE**.

The **setvfsent** subroutine rewinds the **vfs** file to allow repeated searches.

The **endvfsent** subroutine closes the **vfs** file when processing is complete.

Parameters

Item	Description
vfsType	Specifies a **vfs** type.
vfsName	Specifies a **vfs** name.
vfsFlag	Specifies either **VFS_DFLT_LOCAL** or **VFS_DFLT_REMOTE**.

Return Values

The **getvfsent**, **getvfsbytype**, **getvfsbyname**, and **getvfsbyflag** subroutines return a pointer to a **vfs_ent** structure containing the broken-out fields of a line in the **/etc/vfs** file. If an end-of-file character or an error is encountered on reading, a null pointer is returned.

Files

Item	Description
/etc/vfs	Describes the virtual file system (VFS) installed on the system.

getwc, fgetwc, or getwchar Subroutine

Purpose

Gets a wide character from an input stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
wint_t getwc ( Stream)
FILE *Stream;
```

```
wint_t fgetwc (Stream)
FILE *Stream;
```

```
wint_t getwchar (void)
```

Description

The **fgetwc** subroutine obtains the next wide character from the input stream specified by the *Stream* parameter, converts it to the corresponding wide character code, and advances the file position indicator the number of bytes corresponding to the obtained multibyte character. The **getwc** subroutine is equivalent to the **fgetwc** subroutine, except that when implemented as a macro, it may evaluate the *Stream* parameter more than once. The **getwchar** subroutine is equivalent to the **getwc** subroutine with **stdin** (the standard input stream).

The first successful run of the **fgetc**, **fgets**, **fgetwc**, **fgetws**, **fread**, **fscanf**, **getc**, **getchar**, **gets**, or **scanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc** or **ungetwc** subroutine marks the `st_atime` field for update.

Parameters

Item	Description
Stream	Specifies input data.

Return Values

Upon successful completion, the **getwc** and **fgetwc** subroutines return the next wide character from the input stream pointed to by the *Stream* parameter. The **getwchar** subroutine returns the next wide character from the input stream pointed to by stdin.

If the end of the file is reached, an indicator is set and **WEOF** is returned. If a read error occurs, an error indicator is set, **WEOF** is returned, and the **errno** global variable is set to indicate the error.

Error Codes

If the **getwc**, **fgetwc**, or **getwchar** subroutine is unsuccessful because the stream is not buffered or data needs to be read into the buffer, it returns one of the following error codes:

Item	Description
EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter, delaying the process.
EBADF	Indicates that the file descriptor underlying the *Stream* parameter is not valid and cannot be opened for reading.
EINTR	Indicates that the process has received a signal that terminates the read operation.
EIO	Indicates that a physical error has occurred, or the process is in a background process group attempting to read from the controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group is orphaned.
EOVERFLOW	Indicates that the file is a regular file and an attempt has been made to read at or beyond the offset maximum associated with the corresponding stream.

The **getwc**, **fgetwc**, or **getwchar** subroutine is also unsuccessful due to the following error conditions:

Item	Description
ENOMEM	Indicates that storage space is insufficient.
ENXIO	Indicates that the process sent a request to a nonexistent device, or the device cannot handle the request.
EILSEQ	Indicates that the **wc** wide-character code does not correspond to a valid character.

getwd Subroutine

Purpose

Gets current directory path name.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

char *getwd ( PathName)
char *PathName;
```

Description

The **getwd** subroutine determines the absolute path name of the current directory, then copies that path name into the area pointed to by the *PathName* parameter.

The maximum path-name length, in characters, is set by the **PATH_MAX** value, as specified in the **limits.h** file.

Parameters

Item	Description
PathName	Points to the full path name.

Return Values

If the call to the **getwd** subroutine is successful, a pointer to the absolute path name of the current directory is returned. If an error occurs, the **getwd** subroutine returns a null value and places an error message in the *PathName* parameter.

In UNIX03 mode, the getwd subroutine returns a null value if the actual path name is longer than the value defined by PATH_MAX. In the previous mode, the getwd subroutine returns a truncated path name if the path name is longer than PATH_MAX. The previous behavior can be disabled setting the environment variable XPG_SUS_ENV=ON.

getws or fgetws Subroutine

Purpose

Gets a string from a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
wchar_t *fgetws ( WString,  Number,  Stream)
wchar_t *WString;
int Number;
FILE *Stream;
```

```
wchar_t *getws (WString)
wchar_t *WString;
```

Description

The **fgetws** subroutine reads characters from the input stream, converts them to the corresponding wide character codes, and places them in the array pointed to by the *WString* parameter. The subroutine continues until either the number of characters specified by the *Number* parameter minus 1 are read or the subroutine encounters a new-line or end-of-file character. The **fgetws** subroutine terminates the wide character string specified by the *WString* parameter with a null wide character.

The **getws** subroutine reads wide characters from the input stream pointed to by the standard input stream (**stdin**) into the array pointed to by the *WString* parameter. The subroutine continues until it encounters a new-line or the end-of-file character, then it discards any new-line character and places a null wide character after the last character read into the array.

Parameters

Item	Description
WString	Points to a string to receive characters.
Stream	Points to the **FILE** structure of an open file.
Number	Specifies the maximum number of characters to read.

Return Values

If the **getws** or **fgetws** subroutine reaches the end of the file without reading any characters, it transfers no characters to the *String* parameter and returns a null pointer. If a read error occurs, the **getws** or **fgetws** subroutine returns a null pointer and sets the **errno** global variable to indicate the error.

Error Codes

If the **getws** or **fgetws** subroutine is unsuccessful because the stream is not buffered or data needs to be read into the stream's buffer, it returns one or more of the following error codes:

Item	Description
EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter, and the process is delayed in the **fgetws** subroutine.
EBADF	Indicates that the file descriptor specifying the *Stream* parameter is not a read-access file.
EINTR	Indicates that the read operation is terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfer for this file.

Item	Description
EIO	Indicates that insufficient storage space is available.
ENOMEM	Indicates that insufficient storage space is available.
EILSEQ	Indicates that the data read from the input stream does not form a valid character.

getyx Macro

Purpose

Returns the coordinates of the logical cursor in the specified window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

getyx( Window,  Line,  Column)
WINDOW *Window;
int Line, Column;
```

Description

The **getyx** macro returns the coordinates of the logical cursor in the specified window.

Parameters

Item	Description
Window	Identifies the window to get the cursor location from.
Column	Holds the column coordinate of the logical cursor.
Line	Holds the line or row coordinate of the logical cursor.

Example

To get the location of the logical cursor in the user-defined window `my_window` and then put these coordinates in the user-defined integer variables `Line` and `Column`, enter:

```
WINDOW *my_window;
int line, column;
getyx(my_window, line, column);
```

glob Subroutine

Purpose

Generates path names.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <glob.h>

int glob (Pattern, Flags, (Errfunc)(), Pglob)
const char  *Pattern;
int  Flags;
int  *Errfunc (Epath, Eerrno)
const char  *Epath;
int  Eerrno;
glob_t  *Pglob;
```

Description

The **glob** subroutine constructs a list of accessible files that match the *Pattern* parameter.

The **glob** subroutine matches all accessible path names against this pattern and develops a list of all matching path names. To have access to a path name, the **glob** subroutine requires search permission on every component of a path except the last, and read permission on each directory of any file name component of the *Pattern* parameter that contains any of the special characters * (asterisk), ? (question mark), or [(left bracket). The **glob** subroutine stores the number of matched path names and a pointer to a list of pointers to path names in the *Pglob* parameter. The path names are in sort order, based on the setting of the **LC_COLLATE** category in the current locale. The first pointer after the last path name is a null character. If the pattern does not match any path names, the returned number of matched paths is zero.

Parameters

Pattern
Contains the file name pattern to compare against accessible path names.

Flags
Controls the customizable behavior of the **glob** subroutine.

The *Flags* parameter controls the behavior of the **glob** subroutine. The *Flags* value is the bitwise inclusive OR of any of the following constants, which are defined in the **glob.h** file:

GLOB_APPEND
Appends path names located with this call to any path names previously located. If the **GLOB_APPEND** constant is not set, new path names overwrite previous entries in the *Pglob* array. The **GLOB_APPEND** constant should not be set on the first call to the **glob** subroutine. It may, however, be set on subsequent calls.

The **GLOB_APPEND** flag can be used to append a new set of path names to those found in a previous call to the **glob** subroutine. If the **GLOB_APPEND** flag is specified in the *Flags* parameter, the following rules apply:

- If the application sets the **GLOB_DOOFFS** flag in the first call to the **glob** subroutine, it is also set in the second. The value of the *Pglob* parameter is not modified between the calls.
- If the application did not set the **GLOB_DOOFFS** flag in the first call to the **glob** subroutine, it is not set in the second.
- After the second call, the *Pglob* parameter points to a list containing the following:
  - Zero or more null characters, as specified by the **GLOB_DOOFFS** flag.
  - Pointers to the path names that were in the *Pglob* list before the call, in the same order as after the first call to the **glob** subroutine.
  - Pointers to the new path names generated by the second call, in the specified order.
- The count returned in the *Pglob* parameter is the total number of path names from the two calls.
- The application should not modify the *Pglob* parameter between the two calls.

It is the caller's responsibility to create the structure pointed to by the *Pglob* parameter. The **glob** subroutine allocates other space as needed.

GLOB_DOOFFS
Uses the **gl_offs** structure to specify the number of null pointers to add to the beginning of the **gl_pathv** component of the *Pglob* parameter.

GLOB_ERR
Causes the **glob** subroutine to return when it encounters a directory that it cannot open or read. If the **GLOB_ERR** flag is not set, the **glob** subroutine continues to find matches if it encounters a directory that it cannot open or read.

GLOB_MARK
Specifies that each path name that is a directory should have a / (slash) appended.

GLOB_NOCHECK
If the *Pattern* parameter does not match any path name, the **glob** subroutine returns a list consisting only of the *Pattern* parameter, and the number of matched patterns is one.

GLOB_NOSORT
Specifies that the list of path names need not be sorted. If the **GLOB_NOSORT** flag is not set, path names are collated according to the current locale.

GLOB_QUOTE
If the **GLOB_QUOTE** flag is set, a \ (backslash) can be used to escape metacharacters.

Errfunc
Specifies an optional subroutine that, if specified, is called when the **glob** subroutine detects an error condition.

Pglob
Contains a pointer to a **glob_t** structure. The structure is allocated by the caller. The array of structures containing the file names matching the *Pattern* parameter are defined by the **glob** subroutine. The last entry is a null pointer.

Epath
Specifies the path that failed because a directory could not be opened or read.

Eerrno
Specifies the **errno** value of the failure indicated by the *Epath* parameter. This value is set by the **opendir**, **readdir**, or **stat** subroutines.

Return Values

On successful completion, the **glob** subroutine returns a value of 0. The *Pglob* parameter returns the number of matched path names and a pointer to a null-terminated list of matched and sorted path names. If the number of matched path names in the *Pglob* parameter is zero, the pointer in the *Pglob* parameter is undefined.

Error Codes

If the **glob** subroutine terminates due to an error, it returns one of the nonzero constants below. These are defined in the **glob.h** file. In this case, the *Pglob* values are still set as defined in the Return Values section.

Item	Description
GLOB_ABORTED	Indicates the scan was stopped because the **GLOB_ERROR** flag was set or the subroutine specified by the **errfunc** parameter returned a nonzero value.
GLOB_NOSPACE	Indicates a failed attempt to allocate memory.

If, during the search, a directory is encountered that cannot be opened or read and the *Errfunc* parameter is not a null value, the **glob** subroutine calls the subroutine specified by the **errfunc** parameter with two arguments:

- The *Epath* parameter specifies the path that failed.
- The *Eerrno* parameter specifies the value of the **errno** global variable from the failure, as set by the **opendir**, **readdir**, or **stat** subroutine.

If the subroutine specified by the *Errfunc* parameter is called and returns nonzero, or if the **GLOB_ERR** flag is set in the *Flags* parameter, the **glob** subroutine stops the scan and returns **GLOB_ABORTED** after setting the *Pglob* parameter to reflect the paths already scanned. If **GLOB_ERR** is not set and either the *Errfunc* parameter is null or `*errfunc` returns zero, the error is ignored.

The *Pglob* parameter has meaning even if the **glob** subroutine fails. Therefore, the **glob** subroutine can report partial results in the event of an error. However, if the number of matched path names is 0, the pointer in the *Pglob* parameter is unspecified even if the **glob** subroutine did not return an error.

Examples

The **GLOB_NOCHECK** flag can be used with an application to expand any path name using wildcard characters. However, the **GLOB_NOCHECK** flag treats the pattern as just a string by default. The **sh** command can use this facility for option parameters, for example.

The **GLOB_DOOFFS** flag can be used by applications that build an argument list for use with the **execv**, **execve**, or **execvp** subroutine. For example, an application needs to do the equivalent of `ls -l *.c`, but for some reason cannot. The application could still obtain approximately the same result using the sequence:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] ="-l";
execvp ("ls", &globbuf.gl_pathv[0]);
```

Using the same example, `ls -l *.c *.h` could be approximated using the **GLOB_APPEND** flag as follows:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob ("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
```

The new path names generated by a subsequent call with the **GLOB_APPEND** flag set are not sorted together with the previous path names. This is the same way the shell handles path name expansion when multiple expansions are done on a command line.

globfree Subroutine

Purpose

Frees all memory associated with the *pglob* parameter.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <glob.h>
```

void globfree (*pglob*)
glob_t *pglob;

Description

The **globfree** subroutine frees any memory associated with the *pglob* parameter due to a previous call to the **glob** subroutine.

Parameters

Item	Description
pglob	Structure containing the results of a previous call to the **glob** subroutine.

grantpt Subroutine

Purpose

Changes the mode and ownership of a pseudo-terminal device.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

int grantpt ( FileDescriptor)
int FileDescriptor;
```

Description

The **grantpt** subroutine changes the mode and the ownership of the worker pseudo-terminal associated with the controller pseudo-terminal device defined by the *FileDescriptor* parameter. The user ID of the worker pseudo-terminal is set to the real UID of the calling process. The group ID of the worker pseudo-terminal is set to an unspecified group ID. The permission mode of the worker pseudo-terminal is set to readable and writeable by the owner, and writeable by the group.

Parameters

Item	Description
FileDescriptor	Specifies the file descriptor of the controller pseudo-terminal device.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **grantpt** function may fail if:

Item	Description
EBADF	The *fildes* argument is not a valid open file descriptor.
EINVAL	The *fildes* argument is not associated with a controller pseudo-terminal device.
EACCES	The corresponding worker pseudo-terminal device could not be accessed.

h

The following Base Operating System (BOS) runtime services begin with the letter *h*.

halfdelay Subroutine

Purpose

Controls input character delay mode.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int halfdelay(int tenths);
```

Description

The **halfdelay** subroutine sets the input mode for the current window to Half-Delay Mode and specifies tenths of seconds as the half-delay interval. The *tenths* argument must be in a range from 1 up to and including 255.

Flag

Item	Description
x	Instructs **wgetch** to wait *x* tenths of a second for input before timing out.

Parameters

Item	Description
tenths	

Return Values

Upon successful completion, the **halfdelay** subroutine returns OK. Otherwise, it returns ERR.

has_colors Subroutine

Purpose

Determines whether a terminal supports color.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
has_colors()
```

Description

The **has_colors** subroutine determines whether a terminal supports color. If the terminal supports color, the **has_colors** subroutine returns TRUE. Otherwise, it returns FALSE. Because this subroutine tests for color, you can call it before the **start_color** subroutine.

The **has_colors** routine makes writing terminal-independent programs easier because you can use the subroutine to determine whether to use color or another video attribute.

Use the **can_change_colors** subroutine to determine whether a terminal that supports colors also supports changing its color definitions.

Examples

To determine whether or not a terminal supports color, use:

```
has_colors();
```

has_ic and has_il Subroutine

Purpose

Query functions for terminal insert and delete capability.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
bool has_ic(void);
```

```
bool has_il(void);
```

Description

The **has_ic** subroutine indicates whether the terminal has insert- and delete-character capabilities.

The **has_il** subroutine indicates whether the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions.

Return Values

The **has_ic** subroutine returns a value of TRUE if the terminal has insert- and delete-character capabilities. Otherwise, it returns FALSE.

The **has_il** subroutine returns a value of TRUE if the terminal has insert- and delete-line capabilities. Otherwise, it returns FALSE.

Examples

For the **has_ic** subroutine:

To determine the insert capability of a terminal by returning TRUE or FALSE into the user-defined variable `insert_cap`, enter:

```
int insert_cap;
insert_cap = has_ic();
```

For the **has_il** subroutine:

To determine the insert capability of a terminal by returning TRUE or FALSE into the user-defined variable `insert_line`, enter:

```
int insert_line;
insert_line = has_il();
```

has_il Subroutine

Purpose

Determines whether the terminal has insert-line capability.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
has_il( )
```

Description

The **has_il** subroutine determines whether a terminal has insert-line capability.

Return Values

The **has_il** subroutine returns TRUE if terminal has insert-line capability and FALSE, if not.

Examples

To determine the insert capability of a terminal by returning TRUE or FALSE into the user-defined variable `insert_line`, enter:

```
int insert_line;
insert_line = has_il();
```

HBA_CloseAdapter Subroutine

Purpose

Closes the adapter opened by the **HBA_OpenAdapter** subroutine.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

void HBA_CloseAdapter (handle)
HBA_HANDLE handle;
```

Description

The **HBA_CloseAdapter** subroutine closes the file associated with the file handle that was the result of a call to the **HBA_OpenAdapter** subroutine. The **HBA_CloseAdapter** subroutine calls the **close** subroutine, and applies it to the file handle. After performing the operation, the handle is set to NULL.

Parameters

Item	Description
handle	Specifies the open file descriptor obtained from a successful call to the **open** subroutine.

HBA_FreeLibrary Subroutine

Purpose

Frees all the resources allocated to build the Common HBA API Library.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_FreeLibrary ()
```

Description

The **HBA_FreeLibrary** subroutine frees all resources allocated to build the Common HBA API library. This subroutine must be called after calling any other routine from the Common HBA API library.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

Item	Description
HBA_STATUS_OK	A value of 0 on successful completion.
HBA_STATUS_ERROR	A value of 1 if an error occurred.

HBA_GetAdapterAttributes, HBA_GetPortAttributes, HBA_GetDiscoveredPortAttributes, HBA_GetPortAttributesByWWN Subroutine

Purpose

Gets the attributes of the end device's adapter, port, or remote port.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetAdapterAttributes (handle, hbaattributes)
HBA_STATUS HBA_GetAdapterPortAttributes (handle, portindex, portattributes)
HBA_STATUS HBA_GetDiscoveredPortAttributes (handle, portindex, discoveredportindex,
portattributes)
HBA_STATUS HBA_GetPortAttributesByWWN (handle, PortWWN, portattributes)

HBA_HANDLE handle;
HBA_ADAPTERATTRIBUTES *hbaattributes;
HBA_UINT32 portindex;
HBA_PORTATTRIBUTES *portattributes;
HBA_UINT32 discoveredportindex;
HBA_WWN PortWWN;
```

Description

The **HBA_GetAdapterAttributes** subroutine queries the ODM and makes system calls to gather information pertaining to the adapter. This information is returned to the **HBA_ADAPTERATTRIBUTES** structure. This structure is defined in the **/usr/include/sys/hbaapi.h** file.

The **HBA_GetAdapterAttributes**, **HBA_GetAdapterPortAttributes**, **HBA_GetDiscoveredPortAttributes**, and **HBA_GetPortAttributesByWWN** subroutines return the attributes of the adapter, port or remote port.

These attributes include:

- Manufacturer
- SerialNumber
- Model
- ModelDescription
- NodeWWN
- NodeSymbolicName
- HardwareVersion
- DriverVersion
- OptionROMVersion
- FirmwareVersion
- VendorSpecificID
- NumberOfPorts
- Drivername

The **HBA_GetAdapterPortAttributes**, **HBA_GetDiscoveredPortAttributes**, and **HBA_GetPortAttributesByWWN** subroutines also query the ODM and make system calls to gather

information. The gathered information pertains to the port attached to the adapter or discovered on the network. The attributes are stored in the **HBA_PORTATTRIBUTES** structure. This structure is defined in the **/usr/include/sys/hbaapi.h** file.

These attributes include:

- NodeWWN
- PortWWN
- PortFcId
- PortType
- PortState
- PortSupportedClassofService
- PortSupportedFc4Types
- PortActiveFc4Types
- OSDeviceName
- PortSpeed
- NumberofDiscoveredPorts
- PortSymbolicName
- PortSupportedSpeed
- PortMaxFrameSize
- FabricName

The **HBA_GetAdapterPortAttributes** subroutine returns the attributes of the attached port.

The **HBA_GetDiscoveredPortAttributes**, and **HBA_GetPortAttributesByWWN** subroutines return the same information. However, these subroutines differ in the way they are called, and in the way they acquire the information.

Parameters

Item	Description
handle	Specifies the open file descriptor obtained from a successful call to the **open** subroutine.
hbaatributes	Points to an **HBA_AdapterAttributes** structure, which is used to store information pertaining to the Host Bus Adapter.
portindex	Specifies the index number of the port where the information was obtained.
portattributes	Points to an **HBA_PortAttributes** structure used to store information pertaining to the port attached to the Host Bus Adapter.
discoveredportindex	Specifies the index of the attached port discovered over the network.
PortWWN	Specifies the world wide name or port name of the target device.

Return Values

Upon successful completion, the attributes and a value of HBA_STATUS_OK, or 0 are returned.

If no information for a particular attribute is available, a null value is returned for that attribute. HBA_STATUS_ERROR or 1 is returned if certain ODM queries or system calls fail while trying to retrieve the attributes.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

Item	Description
HBA_STATUS_OK	A value of 0 on successful completion.
HBA_STATUS_ERROR	A value of 1 if an error occurred.
HBA_STATUS_ERROR_INVALID_HANDLE	A value of 3 if there was an invalid file handle.
HBA_STATUS_ERROR_ARG	A value of 4 if there was a bad argument.
HBA_STATUS_ERROR_ILLEGAL_WWN	A value of 5 if the world wide name was not recognized.

HBA_GetAdapterName Subroutine

Purpose

Gets the name of a Common Host Bus Adapter.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetAdapterName (adapterindex, adaptername)
HBA_UINT32 adapterindex;
char *adaptername;
```

Description

The **HBA_GetAdapterName** subroutine gets the name of a Common Host Bus Adapter. The *adapterindex* parameter is an index into an internal table containing all FCP adapters on the machine. The *adapterindex* parameter is used to search the table and obtain the adapter name. The name of the adapter is returned in the form of `mgfdomain-model-adapterindex`. The name of the adapter is used as an argument for the **HBA_OpenAdapter** subroutine. From the **HBA_OpenAdapter** subroutine, the file descriptor will be obtained where additional Common HBA API routines can then be called using the file descriptor as the argument.

Parameters

Item	Description
adapterindex	Specifies the index of the adapter held in the adapter table for which the name of the adapter is to be returned.
adaptername	Points to a character string that will be used to hold the name of the adapter.

Return Values

Upon successful completion, the **HBA_GetAdapterName** subroutine returns the name of the adapter and a 0, or a status code of HBA_STATUS_OK. If unsuccessful, a null value will be returned for *adaptername* and an value of 1, or a status code of HBA_STATUS_ERROR.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

Item	Description
HBA_STATUS_OK	A value of 0 on successful completion.
HBA_STATUS_ERROR	A value of 1 if an error occurred.
HBA_STATUS_ERROR_NOT_SUPPORTED	A value of 2 if the function is not supported.
HBA_STATUS_ERROR_INVALID_HANDLE	A value of 3 if there was an invalid file handle.
HBA_STATUS_ERROR_ARG	A value of 4 if there was a bad argument.
HBA_STATUS_ERROR_ILLEGAL_WWN	A value of 5 if the world wide name was not recognized.
HBA_STATUS_ERROR_ILLEGAL_INDEX	A value of 6 if an index was not recognized.
HBA_STATUS_ERROR_MORE_DATA	A value of 7 if a larger buffer is required.
HBA_STATUS_ERROR_STALE_DATA	A value of 8 if information has changed since the last call to the **HBA_RefreshInformation** subroutine.
HBA_STATUS_SCSI_CHECK_CONDITION	A value of 9 if a SCSI Check Condition was reported.
HBA_STATUS_ERROR_BUSY	A value of 10 if the adapter was busy or reserved. Try again later.
HBA_STATUS_ERROR_TRY_AGAIN	A value of 11 if the request timed out. Try again later.
HBA_STATUS_ERROR_UNAVAILABLE	A value of 12 if the referenced HBA has been removed or deactivated.

HBA_GetEventBuffer Subroutine

Purpose

Removes and returns the next events from the HBA's event queue.

Syntax

```
HBA_STATUS HBA_GetEventBuffer(
        HBA_HANDLE handle,
        HBA_EVENTINFO *pEventBuffer,
        HBA_UINT32 *pEventCount,
);
```

Description

The **HBA_GetEventBuffer** function removes and returns the next events from the HBA's event queue. The number of events returned is the lesser of the value of the *EventCount* parameter at the time of the call and the number of entries available in the event queue.

Parameters

Item	Description
handle	A handle to an open HBA.

Item	Description
pEventBuffer	Pointer to a buffer to receive events.
pEventCount	Pointer to the number of event records that fit in the space allocated for the buffer to receive events. It is set to the size (in event records) of the buffer for receiving events on call, and is returned as the number of events actually delivered.

Return Values

The value of the **HBA_GetEventBuffer** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that no errors were encountered and *pEventCount* indicates the number of event records returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pEventBuffer	Remains unchanged. The buffer to which it points contains event records representing previously undelivered events.
pEventCount	Remains unchanged. The value of the integer to which it points contains the number of event records that actually were delivered.

Error Codes

Item	Description
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_GetFC4Statistics Subroutine

Purpose

Returns traffic statistics for a specific FC-4 protocol through a specific local HBA and local end port.

Syntax

```
HBA_STATUS HBA_GetFC4Statistics(
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_UINT8 FC4type,
        HBA_FC4STATISTICS *statistics
);
```

Description

The **HBA_GetFC4Statistics** function returns traffic statistics for a specific FC-4 protocol through a specific local HBA and local end port.

Note: Basic Link Service, Extended Link Service, and CT each have specific Data Structure **TYPE** values, so their traffic can be requested.

Parameters

Item	Description
handle	A handle to an open HBA containing the end port for which FC-4 statistics can return.
hbaPortWWN	The Port Name of the local HBA end port for which FC-4 statistics can return.
FC4type	The Data Structure **TYPE** assigned by FC-FS to the FC-4 protocol for which FC-4 statistics are requested.
statistics	A pointer to an FC-4 Statistics structure in which the statistics for the specified FC-4 protocol can be returned.

Return Values

The value of the **HBA_GetFC4Statistics** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the statistics for the specified FC-4 and end port have been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return value for the following parameter is as follows:

Item	Description
statistics	Remains unchanged. The structure to which it points contains the statistics for the specified FC-4 protocol.

Error Codes

Item	Description
HBA_STATUS_ERROR_ILLEGAL_WWN	Indicates that the HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR_UNSUPPORTED_FC4	Indicates that the specified HBA end port does not support the specified FC-4 protocol.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_GetFcpPersistentBinding Subroutine

Purpose

Gets persistent binding information of SCSI LUNs.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetFcpPersistentBinding (handle, binding)
HBA_HANDLE handle;
PHBA_FCPBinding binding;
```

Description

For the specified HBA_HANDLE, the **HBA_GetFcpPersistentBinding** subroutine returns the full binding information of local SCSI LUNs to FCP LUNs for each child of the specified HBA_HANDLE. Applications must allocate memory for the **HBA_FCPBINDING** structure, and also pass to the subroutine the number of entries allocated. If the subroutine determines that the structure is not large enough to represent the full binding information, it will set the *NumberOfEntries* variable to the correct value and return an error.

Parameters

Item	Description
handle	An HBA_HANDLE to an open adapter.

binding	A pointer to a structure containing the binding information of the handle's children. The **HBA_FCPBINDING** structure has the following form:
	`struct HBA_FCPBinding {`
	`  HBA_UINT32 NumberOfEntries;`
	`  HBA_FCPBINDINGENTRY entry[1]; /* Variable length array */`
	`  };`
	The size of the structure is determined by the calling application, and is passed in by the *NumberOfEntries* variable.

Return Values

Upon successful completion, HBA_STATUS_OK is returned, and the *binding* parameter points to the full binding structure. If the application has not allocated enough space for the full binding, HBA_STATUS_ERROR_MORE_DATA is returned and the *NumberOfEntries* field in the binding structure is set to the correct value.

Error Codes

If there is insufficient space allocated for the full binding. HBA_STATUS_ERROR_MORE_DATA is returned.

HBA_GetFCPStatistics Subroutine

Purpose

Returns traffic statistics for a specific OS SCSI logical unit provided by the FCP protocol on a specific local HBA.

Syntax

```
HBA_STATUS HBA_GetFCPStatistics(
        HBA_HANDLE handle,
        const HBA_SCSIID *lunit,
        HBA_FC4STATISTICS *statistics
);
```

Description

The **HBA_GetFCPStatistics** function returns traffic statistics for a specific OS SCSI logical unit provided by the FCP protocol on a specific local HBA.

Parameters

Item	Description
handle	A handle to an open HBA containing the end port for which FCP-2 statistics can be returned.
lunit	Pointer to a structure specifying the OS SCSI logical unit for which FCP-2 statistics are requested.
statistics	Pointer to a FC-4 Statistics structure in which the FCP-2 statistics for the specified logical unit can be returned.

Return Values

The value of the **HBA_GetFCPStatistics** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that FCP-2 statistics have been returned for the specified HBA. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return value for the following parameter is as follows:

Item	Description
statistics	Remains unchanged. The structure to which it points contains the FCP-2 statistics for the specified HBA and logical unit.

Error Codes

Item	Description
HBA_STATUS_ERROR_INVALID_LUN	The HBA referenced by *handle* does not support the logical unit referenced by *lunit*.
HBA_STATUS_ERROR_UNSUPPORTED_FC4	The specified HBA end port does not support FCP-2.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_GetFcpTargetMappingV2 Subroutine

Purpose

Returns the mapping between OS targets or logical units and FCP targets or logical units offered by the specified HBA end port at the time the function call is processed.

Syntax

```
HBA_STATUS HBA_GetFcpTargetMappingV2(
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_FCPTARGETMAPPINGV2 *pMapping
);
```

Description

The **HBA_GetFcpTargetMappingV2** function returns the mapping between OS identification of SCSI targets or logical units and FCP identification of targets or logical units offered by the specified HBA end port at the time the function call is processed. Space in the *pMapping* structure permitting, one mapping

entry is returned for each FCP logical unit represented in the OS and one mapping entry is returned for each FCP target that is represented in the OS but for which no logical units are represented in the OS. No target mapping entries are returned to represent FCP objects that are not represented in the OS (that is, objects that are unmapped).

The mappings returned include a Logical Unit Unique Device Identifier (LUID) for each logical unit that provides one. For logical units that provide more than one LUID, the LUID returned is the type 3 (FC **Name_Identifier**) LUID with the smallest identifier value if any LUID of type 3 is provided; otherwise, the type 2 (IEEE EUI-64) LUID with the smallest identifier value if any LUID of type 2 is provided; otherwise, the type 1 (T10 vendor identification) LUID with the smallest identifier value if any LUID of type 1 is provided; otherwise, the type 0 (vendor specific) LUID with the smallest identifier value. If the logical unit provides no LUID, the value of the first four bytes of the LUID field are 0.

Parameters

Item	Description
handle	A handle to an open HBA containing the end port for which target mappings are requested.
hbaPortWWN	Port Name of the local HBA end port for which target mappings are requested.
pMapping	Pointer to an **HBA_FCPTARGETMAPPINGV2** structure. The size of this structure shall be limited by the *NumberOfEntries* value within the structure.

Return Values

The value of the **HBA_GetFcpTargetMappingV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that all mapping entries have been returned for the specified end port. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return value for the following parameter is as follows:

Item	Description
pMapping	Remains unchanged. The structure to which it points contains mapping information from OS identifications of SCSI logical units to FCP identifications of logical units for the specified local HBA end port. The number of entries in the structure is the minimum of the number of entries specified at function call or the full mapping. The value of the *NumberOfEntries* field of the returned structure is the total number of mappings the end port has established. This is true even when the function returns an error stating that the buffer is too small to return all of the established mappings. An upper-level application can either allocate a sufficiently large buffer and check this value after a read, or do a read of the *NumberOfEntries* value separately and allocate a new buffer given the size to accommodate the entire mapping structure.

Error Codes

Item	Description
HBA_STATUS_ERROR_MORE_DATA	More space in the buffer is required to contain mapping information.
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR_NOT_SUPPORTED	The HBA referenced by *handle* does not support target mapping.

Item	Description
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_GetFcpTargetMapping Subroutine

Purpose

Gets mapping of OS identification to FCP indentification for each child of the specified HBA_HANDLE.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetFcpTargetMapping (handle, mapping)
HBA_HANDLE handle;
PHBA_FCPTARGETMAPPING mapping;
```

Description

For the specified HBA_HANDLE, the **HBA_GetFcpTargetMapping** subroutine maps OS identification of all its SCSI logical units to their FCP indentification. Applications must allocate memory for the **HBA_FCPTargetMapping** structure, and also pass to the subroutine the number of entries allocated. If the subroutine determines that the structure is not large enough to represent the entire mapping, it will set the *NumberOfEntries* variable to the correct value and return an error.

Parameters

Item	Description
handle	An HBA_HANDLE to an open adapter.

mapping	A pointer to a structure containing a mapping of the handle's children. The **HBA_FCPTARGETMAPPING** structure has the following form:
	`struct HBA_FCPTargetMapping (`
	`HBA_UINT32 NumberOfEntries;`
	`HBA_FCPSCSIENTRY entry[1] /* Variable length array containing mappings */`
	`);`
	The size of the structure is determined by the calling application, and is passed in by the *NumberOfEntries* variable.

Return Values

If successful, HBA_STATUS_OK is returned and the mapping parameter points to the full mapping structure. If the application has not allocated enough space for the full mapping, HBA_STATUS_ERROR_MORE_DATA is returned, and the *NumberOfEntries* field in the mapping structure is set to the correct value.

Error Codes

If there is insufficient space allocated for the full mapping, HBA_STATUS_ERROR_MORE_DATA is returned.

HBA_GetNumberOfAdapters Subroutine

Purpose

Returns the number of adapters discovered on the system.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_UINT32 HBA_GetNumberOfAdapters ()
```

Description

The **HBA_GetNumberOfAdapters** subroutine returns the number of HBAs supported by the library. The value returned is the current number of HBAs and reflects dynamic change of the HBA inventory without requiring a restart of the system, driver, or library.

Return Values

The **HBA_GetNumberOfAdapters** subroutine returns an integer representing the number of adapters on the machine.

HBA_GetPersistentBindingV2 Subroutine

Purpose

Returns persistent bindings between an FCP target and a SCSI ID for a specified HBA end port.

Syntax

```
HBA_STATUS HBA_GetPersistentBindingV2(
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_FCPTARGETMAPPINGV2 *binding
);
```

Description

The **HBA_GetFcpPersistentBindingV2** function returns persistent bindings between an FCP target and a SCSI ID for a specified HBA end port. The binding information can include bindings to Logical Unit Unique Device Identifiers (LUIDs).

Parameters

Item	Description
handle	A handle to an open HBA containing the end port for which persistent binding can be returned.
hbaPortWWN	The Port Name of the local HBA end port for which persistent binding can be returned.

Item	Description
binding	Pointer to an **HBA_FCPBINDING2** structure. The *NumberOfEntries* field in the structure limits the number of entries that are returned.

Return Values

The value of the **HBA_GetPersistentBindingV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that all binding entries have been returned for the specified end port. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return value for the following parameter is as follows:

Item	Description
binding	Remains unchanged. The structure to which it points contains binding information from OS identifications of SCSI logical units to FCP and LUID identifications of logical units for the specified HBA end port. The number of entries in the structure is the minimum of the number of entries specified at function call or the full set of bindings. The *NumberOfEntries* field contains the total number of bindings established by the end port. An application can either call **HBA_GetPersistentBindingV2** with *NumberOfEntries* set to 0 to retrieve the number of entries available, or allocate a sufficiently large buffer to retrieve entries at first call. The **Status** field of each **HBA_FCPBINDINGENTRY2** substructure is 0.

Error Codes

Item	Description
HBA_STATUS_ERROR_MORE_DATA	More space in the buffer is required to contain binding information.
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR_NOT_SUPPORTED	The HBA referenced by *handle* does not support persistent binding.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_GetPortStatistics Subroutine

Purpose

Gets the statistics for a Host Bus Adapter (HBA).

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetPortStatistics (handle, portindex, portstatistics)
HBA_HANDLE handle;
```

```
HBA_UINT32 portindex;
HBA_PORTSTATISTICS *portstatistics;
```

Description

The **HBA_GetPortStatistics** subroutine retrieves the statistics for the specified adapter. Only single-port adapters are supported, and the *portindex* parameter is disregarded. The exact meaning of events being counted for each statistic is vendor specific. The **HBA_PORTSTATISTICS** structure includes the following fields:

- *SecondsSinceLastReset*
- *TxFrames*
- *TxWords*
- *RxFrames*
- *RxWords*
- *LIPCount*
- *NOSCount*
- *ErrorFrames*
- *DumpedFrames*
- *LinkFailureCount*
- *LossOfSyncCount*
- *LossOfSignalCount*
- *PrimitiveSeqProtocolErrCount*
- *InvalidTxWordCount*
- *InvalidCRCCount*

Parameters

Item	Description
handle	HBA_HANDLE to an open adapter.
portindex	Not used.
portstatistics	Pointer to an HBA_PORTSTATISTICS structure.

HBA_GetRNIDMgmtInfo Subroutine

Purpose

Sends a **SCSI GET RNID** command to a remote port of the end device.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_GetRNIDMgmtInfo (handle, pInfo)
HBA_HANDLE handle;
HBA_MGMTINFO *pInfo;
```

Description

The **HBA_SetRNIDMgmtInfo** subroutine sends a **SCSI GET RNID** (Request Node Identification Data) command through a call to **ioctl** with the **SCIOLCHBA** operation as its argument. The *arg* parameter for the **SCIOLCHBA** operation is the address of a **scsi_chba** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file. The *scsi_chba* parameter block allows the caller to select the **GET RNID** command to be sent to the adapter. The **pInfo** structure stores the RNID data returned from **SCIOLCHBA**. The **pInfo** structure is defined in the **/usr/include/sys/hbaapi.h** file. The structure includes:

- wwn
- unittype
- PortId
- NumberOfAttachedNodes
- IPVersion
- UDPort
- IPAddress
- reserved
- TopologyDiscoveryFlags

If successful, the GET RNID data in *pInfo* is returned from the adapter.

Parameters

Item	Description
handle	Specifies the open file descriptor obtained from a successful call to the **open** subroutine.
pInfo	Specifies the structure containing the information to get or set from the **RNID** command

Return Values

Upon successful completion, the **HBA_GetRNIDMgmtInfo** subroutine returns a pointer to a structure containing the data from the **GET RNID** command and a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, a null value is returned along with a value of HBA_STATUS_ERROR, or a value of 1.

Upon successful completion, the **HBA_SetRNIDMgmtInfo** subroutine returns a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, an HBA_STATUS_ERROR value, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

Item	Description
HBA_STATUS_OK	A value of 0 on successful completion.
HBA_STATUS_ERROR	A value of 1 if an error occurred.
HBA_STATUS_ERROR_INVALID_HANDLE	A value of 3 if there was an invalid file handle.

HBA_GetVersion Subroutine

Purpose

Returns the version number of the Common HBA API.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_UINT32 HBA_GetVersion ()
```

Description

The **HBA_GetVersion** subroutine returns the version number representing the release of the Common HBA API.

Return Values

Upon successful completion, the **HBA_GetVersion** subroutine returns an integer value designating the version number of the Common HBA API.

HBA_LoadLibrary Subroutine

Purpose

Loads a vendor specific library from the Common HBA API.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_LoadLibrary ()
```

Description

The **HBA_LoadLibrary** subroutine loads a vendor specific library from the Common HBA API. This library must be called first before calling any other routine from the Common HBA API.

Return Values

The **HBA_LoadLibrary** subroutine returns a value of 0, or **HBA_STATUS_OK**.

HBA_OpenAdapter Subroutine

Purpose

Opens the specified adapter for reading.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_HANDLE HBA_OpenAdapter (adaptername)
char *adaptername;
```

Description

The **HBA_OpenAdapter** subroutine opens the adapter for reading for the purpose of getting it ready for additional calls from other subroutines in the Common HBA API.

The **HBA_OpenAdapter** subroutine allows an application to open a specified HBA device, giving the application access to the device through the HBA_HANDLE return value. The library ensures that all access to this HBA_HANDLE between **HBA_OpenAdapter** and **HBA_CloseAdapter** calls is to the same device.

Parameters

Item	Description
adaptername	Specifies a string that contains the description of the adapter as returned by the **HBA_GetAdapterName** subroutine.

Return Values

If successful, the **HBA_OpenAdapter** subroutine returns an HBA_HANDLE with a value greater than 0. If unsuccessful, the subroutine returns a 0.

HBA_OpenAdapterByWWN Subroutine

Purpose

Attempts to open a handle to the HBA that contains a **Node_Name** or **N_Port_Name** matching the *wwn* argument.

Syntax

```
HBA_STATUS HBA_OpenAdapterByWWN(
        HBA_HANDLE *pHandle,
        HBA_WWN wwn
);
```

Description

The **HBA_OpenAdapterByWWN** function attempts to open a handle to the HBA that contains a **Node_Name** or **N_Port_Name** matching the *wwn* argument. The specified **Name_Identifier** matches the **Node_Name** or **N_Port_Name** of the HBA. Discovered end ports (remote end ports) are *not* checked for a match.

Parameters

Item	Description
pHandle	Pointer to a handle. The value at entry is irrelevant.
wwn	**Name_Identifier** to match the **Node_Name** or **N_Port_Name** of the HBA to open.

Return Values

The value of the **HBA_OpenAdapterByWWN** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the handle contains a valid HBA handle.

The return values for the following parameter is as follows:

Item	Description
pHandle	Remains unchanged. If the open succeeds, the value to which it points is a handle to the requested HBA. On failure, the value is undefined.

Error Codes

Item	Description
HBA_STATUS_ERROR_ILLEGAL_WWN	There is no HBA with a **Node_Name** or **N_Port_Name** that matches *wwn*.
HBA_STATUS_ERROR_AMBIGUOUS_WWN	Multiple HBAs have a matching **Name_Identifier**. This can occur if the **Node_Name**s of multiple HBAs are identical.
HBA_STATUS_ERROR	Returned to indicate any other problem with opening the HBA.

HBA_RefreshInformation Subroutine

Purpose

Refreshes stale information from the Host Bus Adapter.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

void HBA_RefreshInformation (handle)
HBA_HANDLE handle;
```

Description

The **HBA_RefreshInformation** subroutine refreshes stale information from the Host Bus Adapter. This would reflect changes to information obtained from calls to the **HBA_GetAdapterPortAttributes**, or **HBA_GetDiscoveredPortAttributes** subroutine. Once the application calls the **HBA_RefreshInformation** subroutine, it can proceed to the attributes's call to get the new data.

Parameters

Item	Description
handle	Specifies the open file descriptor obtained from a successful call to the **open** subroutine for which the refresh operation is to be performed.

HBA_ScsiInquiryV2 Subroutine

Purpose

Sends a SCSI INQUIRY command to a remote end port.

Syntax

```
HBA_STATUS HBA_ScsiInquiryV2 (
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_WWN discoveredPortWWN,
        HBA_UINT64 fcLUN,
        HBA_UINT8 CDB_Byte1,
        HBA_UINT8 CDB_Byte2,
        void *pRspBuffer,
        HBA_UINT32 *pRspBufferSize,
        HBA_UINT8 *pScsiStatus,
        void *pSenseBuffer,
        HBA_UINT32 *pSenseBufferSize
);
```

Description

The **HBA_ScsiInquiryV2** function sends a **SCSI INQUIRY** command to a remote end port.

A SCSI command is never sent to an end port that does not have SCSI target functionality. If sending a SCSI command causes a SCSI overlapped command condition with a correctly operating target, the command does not get sent. Proper use of tagged commands is an acceptable means of avoiding a SCSI overlapped command condition with targets that support tagged commands.

Parameters

Item	Description
handle	Open HBA through which the **SCSI INQUIRY** command can be issued.
hbaPortWWN	The Port Name for a local HBA end port through which the **SCSI INQUIRY** command can be issued.
discoveredPortWWN	The Port Name for an end port to which the **SCSI INQUIRY** command can be sent.
fcLUN	The SCSI LUN to which the **SCSI INQUIRY** command can be sent.
CDB_Byte1	The second byte of the CDB for the **SCSI INQUIRY** command. This contains control flag bits. At the time this standard was written, the effects of the value of CDB_Byte1 on a SCSI INQUIRY command were as follows: • 0 – Requests the standard SCSI INQUIRY data. • 1 – Requests the vital product data (EVPD) specified by *CDB_Byte2*. • 2 – Requests command support data (**CmdDt**) for the command specified in *CDB_Byte2*. • Other values – Can cause SCSI Check Condition.

Item	Description
CDB_Byte2	The third byte of the CDB for the **SCSI INQUIRY** command. If *CDB_Byte1* is 1, *CDB_Byte2* is the Vital Product Data page code to request. If *CDB_Byte1* is 2, *CDB_Byte2* is the Operation Code of the command support data requested. For other values of *CDB_Byte1*, the value of *CDB_Byte2* is unspecified, and values other than 0 can cause a SCSI Check Condition.
pRspBuffer	A pointer to a buffer to receive the **SCSI INQUIRY** command response.
pRspBufferSize	A pointer to the size in bytes of the buffer to receive the **SCSI INQUIRY** command response.
pScsiStatus	A pointer to a buffer to receive SCSI status.
pSenseBuffer	A pointer to a buffer to receive SCSI sense data.
pSenseBufferSize	A pointer to the size in bytes of the buffer to receive SCSI sense data.

Return Values

The value of the **HBA_ScsiInquiryV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete payload of a reply to the **SCSI INQUIRY** command has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pRspBuffer	Remains unchanged. If the function value is **HBA_STATUS_OK**, the buffer to which it points contains the response to the **SCSI INQUIRY** command.
pRspBufferSize	Remains unchanged. The value of the integer to which it points is the size in bytes of the response returned by the command. This cannot exceed the size passed as an argument at this pointer.
pScsiStatus	Remains unchanged. The value of the byte to which it points is the SCSI status. If the function value is **HBA_STATUS_OK** or **HBA_STATUS_SCSI_CHECK_CONDITION**, the value of the SCSI status can be interpreted based on the SCSI spec. A SCSI status of **HBA_STATUS_OK** indicates that a SCSI response is in the response buffer. A SCSI status of **HBA_STATUS_SCSI_CHECK_CONDITION** indicates that no value is stored in the response, and the sense buffer contains failure information if available.
pSenseBuffer	Remains unchanged. If the function value is **HBA_STATUS_SCSI_CHECK_CONDITION**, the buffer to which it points contains the sense data for the command.
pSenseBufferSize	Remains unchanged. The value of the integer to which it points is the size in bytes of the sense information returned by the command. This cannot exceed the size passed as an argument at this pointer.

Error Codes

Item	Description
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by handle does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR_NOT_A_TARGET	The identified remote end port does not have SCSI Target functionality.

Item	Description
HBA_STATUS_ERROR_TARGET_BUSY	Unable to send the requested command without causing a SCSI overlapped command condition.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_ScsiReadCapacityV2 Subroutine

Purpose

Sends a SCSI READ CAPACITY command to a remote end port.

Syntax

```
HBA_STATUS HBA_ScsiReadCapacityV2(
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_WWN discoveredPortWWN,
        HBA_UINT64 fcLUN,
        void *pRspBuffer,
        HBA_UINT32 *pRspBufferSize,
        HBA_UINT8 *pScsiStatus,
        void *pSenseBuffer,
        HBA_UINT32 *pSenseBufferSize
);
```

Description

The **HBA_ScsiReadCapacityV2** function sends a SCSI READ CAPACITY command to a remote end port.

A SCSI command is never sent to an end port that does not have SCSI target functionality. If sending a SCSI command causes a SCSI overlapped command condition with a correctly operating target, the command will not be sent. Proper use of tagged commands is an acceptable means of avoiding a SCSI overlapped command condition with targets that support tagged commands.

Parameters

Item	Description
handle	A handle to an open HBA through which the SCSI READ CAPACITY command is issued.
hbaPortWWN	The Port Name for a local HBA end port through which the SCSI READ CAPACITY command is issued.
discoveredPortWWN	The Port Name for an end port to which the SCSI READ CAPACITY command is sent.
fcLUN	The SCSI LUN to which the SCSI READ CAPACITY command is sent.
pRspBuffer	Pointer to a buffer to receive the SCSI READ CAPACITY command response.
pRspBufferSize	Pointer to the size in bytes of the buffer to receive the SCSI READ CAPACITY command response.
pScsiStatus	Pointer to a buffer to receive SCSI status.
pSenseBuffer	Pointer to a buffer to receive SCSI sense data.
pSenseBufferSize	Pointer to the size in bytes of the buffer to receive SCSI sense data.

Return Values

The value of the **HBA_ScsiReadCapacityV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete payload of a reply to the SCSI READ CAPACITY command has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pRspBuffer	Remains unchanged. If the function value is **HBA_STATUS_OK**, the buffer to which it points contains the response to the SCSI READ CAPACITY command.
pRspBufferSize	Remains unchanged. The value of the integer to which it points is the size in bytes of the response returned by the command. This cannot exceed the size passed as an argument at this pointer.
pScsiStatus	Remains unchanged. The value of the byte to which it points is the SCSI status. If the function value is **HBA_STATUS_OK** or **HBA_STATUS_SCSI_CHECK_CONDITION**, the value of the SCSI status can be interpreted based on the SCSI spec. A SCSI status of **HBA_STATUS_OK** indicates that a SCSI response is in the response buffer. A SCSI status of **HBA_STATUS_SCSI_CHECK_CONDITION** indicates that no value is stored in the response, and the sense buffer contains failure information if available.
pSenseBuffer	Remains unchanged. If the function value is **HBA_STATUS_SCSI_CHECK_CONDITION**, the buffer to which it points contains the sense data for the command.
pSenseBufferSize	Remains unchanged. The value of the integer to which it points is the size in bytes of the sense information returned by the command. This cannot exceed the size passed as an argument at this pointer.

Error Codes

Item	Description
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR_NOT_A_TARGET	The identified remote end port does not have SCSI Target functionality.
HBA_STATUS_ERROR_TARGET_BUSY	Unable to send the requested command without causing a SCSI overlapped command condition.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_ScsiReportLunsV2 Subroutine

Purpose

Sends a SCSI REPORT LUNS command to Logical Unit Number 0 of a remote end port.

Syntax

```
HBA_STATUS HBA_ScsiReportLUNsV2(
        HBA_HANDLE handle,
```

```
        HBA_WWN hbaPortWWN,
        HBA_WWN discoveredPortWWN,
        void *pRspBuffer,
        HBA_UINT32 *pRspBufferSize,
        HBA_UINT8 *pScsiStatus,
        void *pSenseBuffer,
        HBA_UINT32 *pSenseBufferSize
);
```

Description

The **HBA_ScsiReportLunsV2** function shall send a SCSI REPORT LUNS command to Logical Unit Number 0 of a remote end port.

A SCSI command is never sent to an end port that does not have SCSI target functionality. If sending a SCSI command causes a SCSI overlapped command condition with a correctly operating target, the command will not be sent. Proper use of tagged commands is an acceptable means of avoiding a SCSI overlapped command condition with targets that support tagged commands.

Parameters

Item	Description
handle	A handle to an open HBA through which the SCSI REPORT LUNS command is issued.
hbaPortWWN	The Port Name for a local HBA end port through which the SCSI REPORT LUNS command is issued.
discoveredPortWWN	The Port Name for an end port to which the SCSI REPORT LUNS command is sent.
pRspBuffer	Pointer to a buffer to receive the SCSI REPORT LUNS command response.
pRspBufferSize	Pointer to the size in bytes of the buffer to receive the SCSI REPORT LUNS command response.
pScsiStatus	Pointer to a buffer to receive SCSI status.
pSenseBuffer	Pointer to a buffer to receive SCSI sense data.
pSenseBufferSize	Pointer to the size in bytes of the buffer to receive SCSI sense data.

Return Values

The value of the **HBA_ScsiReportLunsV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete payload of a reply to the SCSI REPORT LUNS command has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pRspBuffer	Remains unchanged. If the function value is **HBA_STATUS_OK**, the buffer to which it points contains the response to the SCSI REPORT LUNS command.
pRspBufferSize	Remains unchanged. The value of the integer to which it points is the size in bytes of the response returned by the command. This cannot exceed the size passed as an argument at this pointer.

Item	Description
pScsiStatus	Remains unchanged. The value of the byte to which it points is the SCSI status. If the function value is **HBA_STATUS_OK** or **HBA_STATUS_SCSI_CHECK_CONDITION**, the value of the SCSI status can be interpreted based on the SCSI spec. A SCSI status of **HBA_STATUS_OK** indicates that a SCSI response is in the response buffer. A SCSI status of **HBA_STATUS_SCSI_CHECK_CONDITION** indicates that no value is stored in the response, and the sense buffer contains failure information if available.
pSenseBuffer	Remains unchanged. If the function value is **HBA_STATUS_SCSI_CHECK_CONDITION**, the buffer to which it points contains the sense data for the command.
pSenseBufferSize	Remains unchanged. The value of the integer to which it points is the size in bytes of the sense information returned by the command. This cannot exceed the size passed as an argument at this pointer.

Error Codes

Item	Description
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR_NOT_A_TARGET	The identified remote end port does not have SCSI Target functionality.
HBA_STATUS_ERROR_TARGET_BUSY	Unable to send the requested command without causing a SCSI overlapped command condition.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_SendCTPassThru Subroutine

Purpose

Sends a CT pass through frame.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendCTPassThru (handle, pReqBuffer, ReqBufferSize, pRspBuffer, RspBufferSize)
HBA_HANDLE handle;
void *pReqBuffer;
HBA_UINT32 ReqBufferSize;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
```

Description

The **HBA_SendCTPassThru** subroutine sends a CT pass through frame to a fabric connected to the specified handle. The CT frame is routed in the fabric according to the *GS_TYPE* field in the CT frame.

Parameters

Item	Description
handle	HBA_HANDLE to an open adapter.
pReqBuffer	Pointer to a buffer that contains the CT request.
ReqBufferSize	Size of the request buffer.
pRspBuffer	Pointer to a buffer that receives the response of the command.
RspBufferSize	Size of the response buffer.

Return Values

If successful, HBA_STATUS_OK is returned, and the *pRspBuffer* parameter points to the CT response.

Error Codes

If the adapter specified by the *handle* parameter is connected to an arbitrated loop, the **HBA_SendCTPassThru** subroutine returns HBA_STATUS_ERROR_NOT_SUPPORTED. This subroutine is only valid when connected to a fabric.

HBA_SendCTPassThruV2 Subroutine

Purpose

Sends a CT request payload.

Syntax

```
HBA_STATUS HBA_SendCTPassThruV2(
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        void *pReqBuffer,
        HBA_UINT32 *ReqBufferSize,
        void *pRspBuffer,
        HBA_UINT32 *pRspBufferSize,
);
```

Description

The **HBA_SendCTPassThruV2** function sends a CT request payload. An HBA should decode this CT_IU request by, routing the CT frame in a fabric according to the **GS_TYPE** field within the CT frame.

Parameters

Item	Description
handle	A handle to an open HBA through which the CT request is issued.
hbaPortWWN	The Port Name for a local HBA **Nx_Port** through which the CT request is issued.
pReqBuffer	Pointer to a buffer containing the full CT payload, including the CT header, to be sent with byte ordering.
ReqBufferSize	The size of the full CT payload, including the CT header, in bytes.
pRSPBuffer	Pointer to a buffer for the CT response.
pRSPBufferSize	Pointer to the size in bytes of the buffer for the CT response payload.

Return Values

The value of the **SendCTPassThruV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete reply to the CT **Passthru** command has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pRspBuffer	Remains unchanged. The buffer to which it points contains the CT response payload, including the CT header received in response to the frame sent, with byte ordering. If the size of the actual response exceeds the size of the response buffer, trailing data is truncated from the response so that the returned data equals the size of the buffer.
pRspBufferSize	Remains unchanged. The value of the integer to which it points is set to the size (in bytes) of the actual response data.

Error Codes

Item	Description
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an **Nx_Port** with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_SendReadCapacity Subroutine

Purpose

Sends a **SCSI READ CAPACITY** command to a Fibre Channel port.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendReadCapacity (handle, portWWN, fcLUN, pRspBuffer, RspBufferSize,
pSenseBuffer,
SenseBufferSize)
HBA_HANDLE handle;
HBA_WWN portWWN;
HBA_UINT64 fcLUN;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
void *pSenseBuffer;
HBA_UINT32 SenseBufferSize;
```

Description

The **HBA_SendReadCapacity** subroutine sends a **SCSI READ CAPACITY** command to the Fibre Channel port connected to the *handle* parameter and specified by the *portWWN* and *fcLUN* parameters.

Parameters

Item	Description
handle	HBA_HANDLE to an open adapter.
portWWN	Port world-wide name of an adapter.
fcLUN	Fibre Channel LUN to send the **SCSI READ CAPACITY** command to.
pRspBuffer	Pointer to a buffer that receives the response of the command.
RspBufferSize	Size of the response buffer.
pSenseBuffer	Pointer to a buffer that receives sense information.
SenseBufferSize	Size of the sense buffer.

Return Values

If successful, HBA_STATUS_OK is returned and the *pRspBuffer* parameter points to the response to the **READ CAPACITY** command. If an error occurs, HBA_STATUS_ERROR is returned.

Error Codes

If the *portWWN* value is not a valid world-wide name connected to the specified handle, HBA_STATUS_ERROR_ILLEGAL_WWN is returned. On any other types of failures, HBA_STATUS_ERROR is returned.

HBA_SendReportLUNs Subroutine

Purpose

Sends a **SCSI REPORT LUNs** command to a remote port of the end device.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendReportLUNs (handle, PortWWN, pRspBuffer, RspBufferSize, pSenseBuffer, SenseBufferSize)
HBA_HANDLE handle;
HBA_WWN PortWWN;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
void *pSenseBuffer;
HBA_UINT32 SenseBufferSize;
```

Description

The **HBA_SendReportLUNs** subroutine sends a **SCSI REPORT LUNs** command through a call to **ioctl** with the **SCIOLCMD** operation as its argument. The *arg* parameter for the **SCIOLCMD** operation is the address of a **scsi_iocmd** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file. The *scsi_iocmd* parameter block allows the caller to select the SCSI and LUN IDs to be queried. The caller also specifies the SCSI command descriptor block area, command length (SCSI command block length), the time-out value for the command, and a *flags* field.

If successful, the report LUNs data is returned in *pRspBuffer*. The returned report LUNs data must be examined to see if the requested LUN exists.

Parameters

Item	Description
handle	Specifies the open file descriptor obtained from a successful call to the **open** subroutine.
PortWWN	Specifies the world wide name or port name of the target device.
pRspBuffer	Points to a buffer containing the requested instruction for a send/read capacity request to an open adapter.
RspBufferSize	Specifies the size of the buffer to the *pRspBuffer* parameter.
pSenseBuffer	Points to a buffer containing the data returned from a send/read capacity request to an open adapter.
SenseBufferSize	Specifies the size of the buffer to the *pSenseBuffer* parameter.

Return Values

Upon successful completion, the **HBA_SendReportLUNs** subroutine returns a buffer in bytes containing the SCSI report of LUNs, a buffer containing the SCSI sense data, and a value of HBA_STATUS_OK, or a value of 0.

If unsuccessful, an empty buffer for the SCSI report of LUNs, a response buffer containing the failure, and a value of HBA_STATUS_ERROR, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

Item	Description
HBA_STATUS_OK	A value of 0 on successful completion.
HBA_STATUS_ERROR	A value of 1 if an error occurred.
HBA_STATUS_ERROR_INVALID_HANDLE	A value of 3 if there was an invalid file handle.
HBA_STATUS_ERROR_ILLEGAL_WWN	A value of 5 if the world wide name was not recognized.
HBA_STATUS_SCSI_CHECK_CONDITION	A value of 9 if a SCSI Check Condition was reported.

HBA_SendRLS Subroutine

Purpose

Issues a Read Link Error Status Block (RLS) Extended Link Service through the specified HBA end port.

Syntax

```
HBA_STATUS HBA_SendRLS (
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_WWN destWWN,
        void *pRspBuffer,
        HBA_UINT32 *pRspBufferSize,
);
```

Description

The **HBA_SendRLS** function issues a Read Link Error Status Block (RLS) Extended Link Service through the specified HBA end port to request a specified remote FC_Port to return the Link Error Status Block associated with the destination Port Name.

Parameters

Item	Description
handle	A handle to an open HBA through which the ELS is sent.
hbaPortWWN	Port Name of the local HBA end port through which the ELS is sent.
destWWN	Port Name of the remote FC_Port to which the ELS is sent.
pRspBuffer	Pointer to a buffer to receive the ELS response.
pRSPBufferSize	Pointer to the size in bytes of *pRspBuffer*. A size of 28 is sufficient for the largest response.

Return Values

The value of the **HBA_SendRLS** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete LS_ACC to the RLS ELS has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pRspBuffer	Remains unchanged. The buffer to which it points contains the payload data from the RLS Reply. Note that if the ELS was rejected, this is the LS_RJT payload. If the size of the reply payload exceeds the size specified in the *pRspBufferSize* parameter at entry to the function, the returned data is truncated to the size specified in the argument.
pRspBufferSize	Remains unchanged. The value of the integer to which it points contains the size in bytes of the complete ELS reply payload. This can exceed the size specified as an argument. This indicates that the data in *pRspBuffer* has been truncated.

Error Codes

Item	Description
HBA_STATUS_ERROR_ELS_REJECT	The RNID ELS was rejected by the destination FC_Port.
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_SendRNID Subroutine

Purpose

Sends an RNID command through a call to **SCIOLPAYLD** to a remote port of the end device.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendRNID (handle, wwn, wwntype, pRspBuffer, RspBufferSize)
HBA_HANDLE handle;
HBA_WWN wwn;
HBA_WWNTYPE wwntype;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
```

Description

The **HBA_SendRNID** subroutine sends a **SCSI RNID** command with the Node Identification Data Format set to indicate the default Topology Discovery format. This is done through a call to **ioctl** with the **SCIOLPAYLD** operation as its argument. The *arg* parameter for the **SCIOLPAYLD** operation is the address of an **scsi_trans_payld** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file. The *scsi_trans_payld* parameter block allows the caller to select the SCSI and LUN IDs to be queried. In addition, the caller must specify the **fcph_rnid_payld_t** structure to hold the command and the topology format for **SCIOLPAYLD**. The structure for the **fcph_rnid_payld_t** structure is defined in the **/usr/include/sys/fcph.h** file.

If successful, the RNID data is returned in *pRspBuffer*. The returned RNID data must be examined to see if the requested information exists.

Parameters

Item	Description
handle	Specifies the open file descriptor obtained from a successful call to the **open** subroutine.
wwn	Specifies the world wide name or port name of the target device.
wwntype	Specifies the type of the world wide name or port name of the target device.
pRspBuffer	Points to a buffer containing the requested instruction for a send/read capacity request to an open adapter.
RspBufferSize	Specifies the size of the buffer to the *pRspBuffer* parameter.

Return Values

Upon successful completion, the **HBA_SendRNID** subroutine returns a buffer in bytes containing the SCSI RNID data and a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, an empty buffer for the SCSI RNID and a value of HBA_STATUS_ERROR, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

Item	Description
HBA_STATUS_OK	A value of 0 on successful completion.
HBA_STATUS_ERROR	A value of 1 if an error occurred.
HBA_STATUS_ERROR_NOT_SUPPORTED	A value of 2 if the function is not supported.
HBA_STATUS_ERROR_INVALID_HANDLE	A value of 3 if there was an invalid file handle.
HBA_STATUS_ERROR_ILLEGAL_WWN	A value of 5 if the world wide name was not recognized.

HBA_SendRNIDV2 Subroutine

Purpose

Issues an RNID ELS to another FC_Port requesting a specified Node Identification Data Format.

Syntax

```
HBA_STATUS HBA_SendRNIDV2(
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_WWN destWWN,
        HBA_UINT32 destFCID,
        HBA_UINT32 NodeIdDataFormat,
        void *pRspBuffer,
        HBA_UINT32 *pRspBufferSize,
);
```

Description

The **HBA_SendRNIDV2** function issues an RNID ELS to another FC_Port requesting a specified Node Identification Data Format.

The *destFCID* parameter can be set to allow the RNID ELS to be sent to an FC_Port that might not be registered with the name server. If *destFCID* is set to x'00 00 00', the parameter is ignored. If *destFCID* is not 0, the HBA API library verifies that the *destWWN*/*destFCID* pair match in order to limit visibility that can violate scoping mechanisms (such as soft zoning):

- If the *destWWN*/*destFCID* pair matches an entry in the discovered ports table, the RNID is sent.
- If there is no entry in the discovered ports table for the *destWWN* or *destFCID*, the RNID is sent.
- If there is an entry in the discovered ports table for the *destWWN*, but the *destFCID* does not match, then the request is rejected.
- On completion of the **HBA_SendRNIDV2**, if the Common Identification Data Length is nonzero in the RNID response, the API library compares the **N_Port_Name** in the Common Identification Data of the RNID response with *destWWN* and fails the operation without returning the response data if they do not match. If the Common Identification Data Length is 0 in the RNID response, this test is omitted.

Parameters

Item	Description
handle	A handle to an open HBA through which the ELS is sent.
hbaPortWWN	Port Name of the local HBA end port through which the ELS is sent.
destWWN	Port Name of the remote FC_Port to which the ELS is sent.
destFCID	Address identifier of the destination to which the ELS is sent if *destFCID* is nonzero. *destFCID* is ignored if *destFCID* is 0.
NodeIdDataFormat	Valid value for Node Identification Data Format.
pRSPBuffer	Pointer to a buffer to receive the ELS response.
pRSPBufferSize	Pointer to the size in bytes of *pRspBuffer*.

Return Values

The value of the **HBA_SendRNIDV2** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete LS_ACC

to the RNID ELS has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pRspBuffer	Remains unchanged. The buffer to which it points contains the payload data from the RNID Reply. Note that if the ELS was rejected, this is the LS_RJT payload. If the size of the reply payload exceeds the size specified in the *pRspBufferSize* parameter at entry to the function, the returned data is truncated to the size specified in the argument.
pRspBufferSize	Remains unchanged. The value of the integer to which it points contains the size in bytes of the complete ELS reply payload. This can exceed the size specified as an argument. This indicates that the data in *pRspBuffer* has been truncated.

Error Codes

Item	Description
HBA_STATUS_ERROR_ELS_REJECT	The RNID ELS was rejected by the destination end port.
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR_ILLEGAL_FCID	The *destWWN*/*destFCID* pair conflicts with a discovered Port Name/address identifier pair known by the HBA referenced by *handle*.
HBA_STATUS_ERROR_ILLEGAL_FCID	The **N_Port_Name** in the RNID response does not match the *destWWN*.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_SendRPL Subroutine

Purpose

Issues a Read Port List (RPL) Extended Link Service through the specified HBA to a specified end port or domain controller.

Syntax

```
HBA_STATUS HBA_SendRPL (
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_WWN agent_wwn,
        HBA_UINT32 agent_domain,
        HBA_UINT32 portIndex,
        void *pRspBuffer,
        HBA_UINT32 *pRspBufferSize,
);
```

Description

The **HBA_SendRPL** function issues a Read Port List (RPL) Extended Link Service through the specified HBA to a specified end port or domain controller.

Parameters

Item	Description
handle	A handle to an open HBA through which the ELS is sent.
hbaPortWWN	Port Name of the local HBA end port through which the ELS is sent.
agent_wwn	Port Name of an FC_Port that is requested to provide its list of FC_Ports if *agent_wwn* is nonzero. If *agent_wwn* is 0, it is ignored.
agent_domain	Domain number and the domain controller for that domain shall be the entity that shall be requested to provide its list of FC_Ports if *agent_wwn* is 0. If *agent_wwn* is nonzero, *agent_domain* is ignored.
portIndex	Index of the first FC_Port requested in the response list. **Note:** If the recipient has proper compliance, the index of the first FC_Port in the complete list maintained by the recipient of the request is 0.
pRSPBuffer	Pointer to a buffer to receive the ELS response.
pRSPBufferSize	Pointer to the size in bytes of *pRspBuffer*. **Note:** If the responding entity has proper compliance, it truncates the list in the response to the number of FC_Ports that fit.

Return Values

The value of the **HBA_SendRPL** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete LS_ACC to the RPL ELS has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pRspBuffer	Remains unchanged. The buffer to which it points contains the payload data from the RPL Reply. If the ELS was rejected, this is the LS_RJT payload. If the size of the reply payload exceeds the size specified in the *pRspBufferSize* parameter at entry to the function, the returned data is truncated to the size specified in the argument.
pRspBufferSize	Remains unchanged. The value of the integer to which it points contains the size in bytes of the complete ELS reply payload. This can exceed the size specified as an argument. This indicates that the data in *pRspBuffer* has been truncated. **Note:** Truncation is not necessary if the responding entity is of proper compliance.

Error Codes

Item	Description
HBA_STATUS_ERROR_ELS_REJECT	The RPL ELS was rejected by the destination end port.
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.

Item	Description
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_SendRPS Subroutine

Purpose

Issues a Read Port Status Block (RPS) Extended Link Service through the specified HBA to a specified FC_Port or domain controller.

Syntax

```
HBA_STATUS HBA_SendRPS (
        HBA_HANDLE handle,
        HBA_WWN hbaPortWWN,
        HBA_WWN agent_wwn,
        HBA_UINT32 agent_domain,
        HBA_WWN object_wwn,
        HBA_UINT32 object_port_number,
        void *pRspBuffer,
        HBA_UINT32 *pRspBufferSize,
);
```

Description

The **HBA_SendRPS** function issues a Read Port Status Block (RPS) Extended Link Service through the specified HBA to a specified FC_Port or domain controller.

Parameters

Item	Description
handle	A handle to an open HBA through which the ELS is sent.
hbaPortWWN	Port Name of the local HBA end port through which the ELS is sent.
agent_wwn	Port Name of an FC_Port that is requested to provide Port Status if *agent_wwn* is nonzero. *agent_wwn* is ignored if its value is 0.
agent_domain	Domain number for the domain controller that is requested to provide Port status if *agent_wwn* is 0. *agent_domain* is ignored if *agent_wwn* is nonzero.
object_wwn	Port Name of an FC_Port for which Port Status is returned if *object_wwn* is nonzero. *object_wwn* is ignored if its value is 0.
object_port_number	Relative port number of the FC_Port for which Port Status is returned if *object_wwn* is 0. The relative port number is defined in a vendor-specific manner within the entity to which the request is sent. *object_port_number* is ignored if *object_wwn* is nonzero.
pRspBuffer	Pointer to a buffer to receive the ELS response.
pRSPBufferSize	Pointer to the size in bytes of *pRspBuffer*. A size of 56 is sufficient for the largest response.

Return Values

The value of the **HBA_SendRPS** function is a valid status return value that indicates the reason for completion of the requested function. **HBA_STATUS_OK** is returned to indicate that the complete LS_ACC

to the RPS ELS has been returned. A valid status return value that most closely describes the result of the function should be returned to indicate a reason with no required value.

The return values for the following parameters are as follows:

Item	Description
pRspBuffer	Remains unchanged. The buffer to which it points contains the payload data from the RPS Reply. If the ELS was rejected, this is the LS_RJT payload. If the size of the reply payload exceeds the size specified in the *pRspBufferSize* parameter at entry to the function, the returned data is truncated to the size specified in the argument.
pRspBufferSize	Remains unchanged. The value of the integer to which it points contains the size in bytes of the complete ELS reply payload. This can exceed the size specified as an argument. This indicates that the data in *pRspBuffer* has been truncated.

Error Codes

Item	Description
HBA_STATUS_ERROR_ELS_REJECT	The RPS ELS was rejected by the destination end port.
HBA_STATUS_ERROR_ILLEGAL_WWN	The HBA referenced by *handle* does not contain an end port with Port Name *hbaPortWWN*.
HBA_STATUS_ERROR	Returned to indicate any problem with no required value.

HBA_SendScsiInquiry Subroutine

Purpose

Sends a SCSI device inquiry command to a remote port of the end device.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_SendScsiInquiry (handle, PortWWN, fcLUN, EVPD, PageCode, pRspBuffer, RspBufferSize,
pSenseBuffer,
SenseBufferSize)
HBA_HANDLE handle;
HBA_WWN PortWWN;
HBA_UINT64 fcLUN;
HBA_UINT8 EVPD;
HBA_UINT32 PageCode;
void *pRspBuffer;
HBA_UINT32 RspBufferSize;
void *pSenseBuffer;
HBA_UINT32 SenseBufferSize;
```

Description

The **HBA_SendScsiInquiry** subroutine sends a **SCSI INQUIRY** command through a call to **ioctl** with the **SCIOLINQU** operation as its argument. The *arg* parameter for the **SCIOLINQU** operation is the address of an **scsi_inquiry** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file. The *scsi_inquiry* parameter block allows the caller to select the SCSI and LUN IDs to be queried. If

successful, the inquiry data is returned in the *pRspBuffer* parameter. Successful completion occurs if a device responds at the requested SCSI ID, but the returned inquiry data must be examined to see if the requested LUN exists.

Parameters

Item	Description
handle	Specifies the open file descriptor obtained from a successful call to the **open** subroutine.
PortWWN	Specifies the world wide name or port name of the target device.
fcLUN	Specifies the fcLUN.
EVPD	Specifies the value for the EVPD bit. If the value is 1, the Vital Product Data page code will be specified by the *PageCode* parameter.
PageCode	Specifies the Vital Product Data that is to be requested if the EVPD parameter is set to 1.
pRspBuffer	Points to a buffer containing the requested instruction for a send/read capacity request to an open adapter. The size of this buffer must not be greater than 255 bytes.
RspBufferSize	Specifies the size of the buffer to the *pRspBuffer* parameter.
pSenseBuffer	Points to a buffer containing the data returned from a send/read capacity request to an open adapter.
SenseBufferSize	Specifies the size of the buffer to the *pSenseBuffer* parameter.

Return Values

Upon successful completion, the **HBA_SendScsiInquiry** subroutine returns a buffer in bytes containing the SCSI inquiry, a buffer containing the SCSI sense data, and a value of HBA_STATUS_OK, or a value of 0.

If unsuccessful, an empty buffer for the SCSI inquiry, a response buffer containing the failure, and a value of HBA_STATUS_ERROR, or a value of 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

Item	Description
HBA_STATUS_OK	A value of 0 on successful completion.
HBA_STATUS_ERROR	A value of 1 if an error occurred.
HBA_STATUS_ERROR_INVALID_HANDLE	A value of 3 if there was an invalid file handle.
HBA_STATUS_ERROR_ARG	A value of 4 if there was a bad argument.
HBA_STATUS_ERROR_ILLEGAL_WWN	A value of 5 if the world wide name was not recognized.
HBA_STATUS_SCSI_CHECK_CONDITION	A value of 9 if a SCSI Check Condition was reported.

HBA_SetRNIDMgmtInfo Subroutine

Purpose

Sends a **SCSI SET RNID** command to a remote port of the end device.

Library

Common Host Bus Adapter Library (**libHBAAPI.a**)

Syntax

```
#include <sys/hbaapi.h>

HBA_STATUS HBA_SetRNIDMgmtInfo (handle, info)
HBA_HANDLE handle;
HBA_MGMTINFO info;
```

Description

The **HBA_SetRNIDMgmtInfo** subroutine sends a **SCSI SET RNID** (Request Node Identification Data) command with the **SCIOLCHBA** operation as its argument. This is done through a call to **ioctl**. The *arg* parameter for the **SCIOLCHBA** operation is the address of a **scsi_chba** structure. This structure is defined in the **/usr/include/sys/scsi_buf.h** file. The *scsi_chba* parameter block allows the caller to select the **SET RNID** command to be sent to the adapter. The **info** structure stores the RNID data to be set. The **info** structure is defined in the **/usr/include/sys/hbaapi.h** file. The structure includes:

- wwn
- unittype
- PortId
- NumberOfAttachedNodes
- IPVersion
- UDPort
- IPAddress
- reserved
- TopologyDiscoveryFlags

If successful, the SET RNID data in **info** is sent to the adapter.

Parameters

Item	Description
handle	Specifies the open file descriptor obtained from a successful call to the **open** subroutine.
info	Specifies the structure containing the information to be set or received from the **RNID** command

Return Values

Upon successful completion, the **HBA_SetRNIDMgmtInfo** subroutine returns a value of HBA_STATUS_OK, or a value of 0. If unsuccessful, a value of HBA_STATUS_ERROR, or a 1 is returned.

Error Codes

The Storage Area Network Host Bus Adapter API subroutines return the following error codes:

Item	Description
HBA_STATUS_OK	A value of 0 on successful completion.
HBA_STATUS_ERROR	A value of 1 if an error occurred.
HBA_STATUS_ERROR_INVALID_HANDLE	A value of 3 if there was an invalid file handle.

hpmInit, f_hpminit, hpmStart, f_hpmstart, hpmStop, f_hpmstop, hpmTstart, f_hpmtstart, hpmTstop, f_hpmtstop, hpmGetTimeAndCounters, f_hpmgettimeandcounters, hpmGetCounters, f_hpmgetcounters, hpmTerminate, and f_hpmterminate Subroutine

Purpose

Provides application instrumentation for performance monitoring.

Library

HPM Library (`libhpm.a`)

HPM Library (`libhpm.a`) includes four additional subroutines for threaded applications.

Syntax

```
#include <libhpm.h>

void hpmInit(int  taskID, char  *progName);
void f_hpminit(int  taskID, char  *progName);

void hpmStart(int  instID, char  *label);
void f_hpmstart(int  instID, char  *label);

void hpmStop(int  instID);
void f_hpmstop(int  instID);

(libhpm_r only)
void hpmTstart(int  instID, char  *label);
void f_hpmtstart(int  instID, char  *label);
(libhpm_r only)
void hpmTstop(int  instID);
void f_hpmtstop(int  instID);

void hpmGetTimeAndCounters(int  numCounters, double  *time, long long  *values);
void f_hpmgettimeandcounters(int  numCounters, double  *time, long long  *values);

void hpmGetCounters(long long  *values);
void f_hpmgetcounters(long long  *values);

void hpmTerminate(int  taskID);
void f_hpmterminate(int  taskID);
```

Description

The `hpmInit` and `f_hpminit` subroutines initialize tasks specified by the *taskID* and *progName* parameters.

The `hpmStart` and `f_hpmstart` subroutines debut an instrumented code segment. If more than 100 instrumented sections are required, the `HPM_NUM_INST_PTS` environment variable can be set to indicate the higher value and *instID* should be less than this value.

The `hpmStop` and `f_hpmstop` subroutines indicate the end of the instrumented code segment *instID*. For each call to `hpmStart` and `f_hpmstart`, there should be a corresponding call to `hpmStop` and `f_hpmstop` with the matching *instID*.

The `hpmTstart` and `f_hpmtstart` subroutines perform the same function as `hpmStart` and `f_hpmstart`, but are used in threaded applications.

The hpmTstop and f_hpmtstop subroutines perform the same function as hpmStop and f_hpmstop, but are used in threaded applications.

The hpmGetTimeAndCounters and f_hpmgettimeandcounters subroutines are used to return the time in seconds and the accumulated counts since the call to hpmInit or f_hpminit.

The hpmGetCounters and f_hpmgetcounters subroutines return all the accumulated counts since the call to hpmInit or f_hpminit. To minimize intrusion and overhead, the hpmGetCounters and f_hpmgetcounters subroutines do not perform any check on the size of the *values* array. The number of counters can be obtained from the pm_info2_t.maxpmcs structure element supplied by pm_initialize or by using the pmlist -s command. Alternatively, the application can use the current maximum value of 8.

The hpmTerminate and f_hpmterminate subroutines end the *taskID* and generate the output. Applications that do not call hpmTerminate or f_hpmterminate, do not generate performance information.

A summary report for each task is written by default in the *progName_pid_taskID*.hpm file, where *progName* is the second parameter to the hpmInit subroutine. If *progName* contains a space or tab character, or is otherwise invalid, a diagnostic message is written to stderr and the library exits with an error to avoid further problems.

The output file name can be defined with the HPM_OUTPUT_NAME environment flag. The libhpm still adds the file name suffix *_taskID*.hpm for the performance files. By using this environment variable, you can specify the date and time for the output file name. For example:

```
MYDATE=$(date +"m%d:11/15/18M%S")
export HPM_OUTPUT_NAME=myprogram_$MYDATE
```

where the output file for task 27 will have the following name:

```
myprogram_yyyymmdd:HHMMSS_0027.hpm
```

The GUI and .viz output is deactivated by default. The aligned set of performance files named *progName_pid_taskID*.viz or HPM_OUTPUT_NAME_*taskID*.viz will not be generated (the generation of the .viz file was previously activated by default and avoided with the HPM_VIZ_OUTPUT = FALSE environment variable).

Parameters

Item	Description
instID	Specifies the instrumented section ID as an integer value greater than 0 and less than 100.
label	Specifies a label with a character string.
numCounters	Specifies an integer value that indicates the number of counters to be accessed.
progName	Specifies a program name using a character string label.
taskID	Specifies a node ID with an integer value.
time	Specifies a double precision float.
values	Specifies an array of type long long of size *numCounters*.

Execution Environment

Functionality provided by the libhpm library is dependent upon corresponding functions in the libpmapi and libm libraries. Therefore, the -lpmapi -lm link flags must be specified when compiling applications.

Return Values

No return values are defined.

Error Codes

Upon failure, these `libhpm` subroutines either write error messages explicitly to `stderr` or use the PMAPI `pm_error` function. The `pm_error` function is called following an error return from any of the following subroutines:

- `pm_init_private`
- `pm_set_program_mygroup`
- `pm_stop_mygroup`
- `pm_get_data_mygroup`
- `pm_start_mygroup`
- `pm_stop_mythread`
- `pm_get_data_mythread`
- `pm_start_mythread`
- `pm_get_data_mythread`

Diagnostic messages are explicitly written to `stderr` or `stdout` in the following situations:

- `pm_cycles` or `gettimeofday` returns an error
- The value of the *instID* parameter is invalid
- An event set is out of range
- The `libHPMevents` file or `HPM_flags.env` file has an incorrect format
- There are internal errors.

Error messages that are not fatal are written to `stdout` or `stderr` with the text `WARNING`.

hsearch, hcreate, or hdestroy Subroutine

Purpose

Manages hash tables.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <search.h>

ENTRY *hsearch ( Item,  Action)
ENTRY Item;
Action Action;

int hcreate ( NumberOfElements)
size_t NumberOfElements;
void hdestroy ( )
```

Description

Attention: Do not use the **hsearch**, **hcreate**, or **hdestroy** subroutine in a multithreaded environment.

The **hsearch** subroutine searches a hash table. It returns a pointer into a hash table that indicates the location of the given item. The **hsearch** subroutine uses open addressing with a multiplicative hash function.

The **hcreate** subroutine allocates sufficient space for the table. You must call the **hcreate** subroutine before calling the **hsearch** subroutine. The *NumberOfElements* parameter is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The **hdestroy** subroutine deletes the hash table. This action allows you to start a new hash table since only one table can be active at a time. After the call to the **hdestroy** subroutine, the data can no longer be considered accessible.

Parameters

Item	Description
Item	Identifies a structure of the type **ENTRY** as defined in the **search.h** file. It contains two pointers: **Item.key** Points to the comparison key. The key field is of the **char** type. **Item.data** Points to any other data associated with that key. The data field is of the **void** type. Pointers to data types other than the **char** type should be declared to pointer-to-character.
Action	Specifies the value of the *Action* enumeration parameter that indicates what is to be done with an entry if it cannot be found in the table. Values are: **ENTER** Enters the value of the *Item* parameter into the table at the appropriate point. If the table is full, the **hsearch** subroutine returns a null pointer. **FIND** Does not enter the value of the *Item* parameter into the table. If the value of the *Item* parameter cannot be found, the **hsearch** subroutine returns a null pointer. If the value of the *Item* parameter is found, the subroutine returns the address of the item in the hash table.
NumberOfElements	Provides an estimate of the maximum number of entries that the table contains. Under some circumstances, the **hcreate** subroutine may actually make the table larger than specified.

Return Values

The **hcreate** subroutine returns a value of 0 if it cannot allocate sufficient space for the table.

hypot, hypotf, hypotl, hypotd32, hypotd64, and hypotd128 Subroutines

Purpose

Computes the Euclidean distance function and complex absolute value.

Libraries

IEEE Math Library (**libm.a**) System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
```

double hypot (*x*, *y*)
double *x*, *y*;

```
float hypotf (x, y)
float x;
float y;

long double hypotl (x, y)
long double x;
long double y;
_Decimal32 hypotd32 (x, y)
_Decimal32 x, y;

_Decimal64 hypotd64 (x, y)
_Decimal64 x, y;

_Decimal128 hypotd128 (x, y)
_Decimal128 x, y;
```

Description

The **hypot**, **hypotf**, **hypotl**, **hypotd32**, **hypotd64**, and **hypotd128** subroutines compute the value of the square root of $x^2 + y^2$ without undue overflow or underflow.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies some double-precision floating-point value.
y	Specifies some double-precision floating-point value.

Return Values

Upon successful completion, the **hypot**, **hypotf**, **hypotl**, **hypotd32**, **hypotd64**, and **hypotd128** subroutines return the length of the hypotenuse of a right-angled triangle with sides of length *x* and *y*.

If the correct value would cause overflow, a range error occurs and the **hypotf**, **hypotl**, **hypotd32**, **hypotd64**, and **hypotd128** subroutines return the value of the macro **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

If *x* or *y* is ±Inf, +Inf is returned (even if one of *x* or *y* is NaN).

If *x* or *y* is NaN, and the other is not ±Inf, a NaN is returned.

If both arguments are subnormal and the correct result is subnormal, a range error may occur and the correct result is returned.

Error Codes

When using the **libm.a** (**-lm**) library, if the correct value overflows, the **hypot** subroutine returns a **HUGE_VAL** value.

Note: (hypot (INF, *value*) and **hypot (***value***,** INF**)** are both equal to **+INF** for all values, even if *value* = NaN.

When using **libmsaa.a** (**-lmsaa**), if the correct value overflows, the **hypot** subroutine returns **HUGE_VAL** and sets the global variable **errno** to **ERANGE**.

These error-handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** (**-lmsaa**) library.

i

The following Base Operating System (BOS) runtime services begin with the letter *i*.

iconv Subroutine

Purpose

Converts a string of characters in one character code set to another character code set.

Library

The **iconv** Library (**libiconv.a**)

Syntax

```
#include <iconv.h>
```

```
size_t iconv (CD, InBuf, InBytesLeft, OutBuf, OutBytesLeft)
iconv_t CD;
char **OutBuf, **InBuf;
size_t *OutBytesLeft, *InBytesLeft;
```

Description

The **iconv** subroutine converts the string specified by the *InBuf* parameter into a different code set and returns the results in the *OutBuf* parameter. The required conversion method is identified by the *CD* parameter, which must be valid conversion descriptor returned by a previous, successful call to the **iconv_open** subroutine.

On calling, the *InBytesLeft* parameter indicates the number of bytes in the *InBuf* buffer to be converted, and the *OutBytesLeft* parameter indicates the number of bytes remaining in the *OutBuf* buffer that do not contain converted data. These values are updated upon return so they indicate the new state of their associated buffers.

For state-dependent encodings, calling the **iconv** subroutine with the *InBuf* buffer set to null will reset the conversion descriptor in the *CD* parameter to its initial state. Subsequent calls with the *InBuf* buffer, specifying other than a null pointer, may cause the internal state of the subroutine to be altered a necessary.

Parameters

Item	Description
CD	Specifies the conversion descriptor that points to the correct code set converter.
InBuf	Points to a buffer that contains the number of bytes in the *InBytesLeft* parameter to be converted.
InBytesLeft	Points to an integer that contains the number of bytes in the *InBuf* parameter.
OutBuf	Points to a buffer that contains the number of bytes in the *OutBytesLeft* parameter that has been converted.
OutBytesLeft	Points to an integer that contains the number of bytes in the *OutBuf* parameter.

Return Values

Upon successful conversion of all the characters in the *InBuf* buffer and after placing the converted characters in the *OutBuf* buffer, the **iconv** subroutine returns 0, updates the *InBytesLeft* and *OutBytesLeft* parameters, and increments the *InBuf* and *OutBuf* pointers. Otherwise, it updates the varibles pointed to by the parameters to indicate the extent to the conversion, returns the number of bytes still left to be converted in the input buffer, and sets the **errno** global variable to indicate the error.

Error Codes

If the **iconv** subroutine is unsuccessful, it updates the variables to reflect the extent of the conversion before it stopped and sets the **errno** global variable to one of the following values:

Item	Description
EILSEQ	Indicates an unusable character. If an input character does not belong to the input code set, no conversion is attempted on the unusable on the character. In *InBytesLeft* parameters indicates the bytes left to be converted, including the first byte of the unusable character. *InBuf* parameter points to the first byte of the unusable character sequence. The values of *OutBuf* and *OutBytesLeft* are updated according to the number of bytes available in the output buffer that do not contain converted data.
E2BIG	Indicates an output buffer overflow. If the *OutBuf* buffer is too small to contain all the converted characters, the character that causes the overflow is not converted. The *InBytesLeft* parameter indicates the bytes left to be converted (including the character that caused the overflow). The *InBuf* parameter points to the first byte of the characters left to convert.
EINVAL	Indicates the input buffer was truncated. If the original value of *InBytesLeft* is exhausted in the middle of a character conversion or shift/lock block, the *InBytesLeft* parameter indicates the number of bytes undefined in the character being converted. If an input character of shift sequence is truncated by the *InBuf* buffer, no conversion is attempted on the truncated data, and the *InBytesLeft* parameter indicates the bytes left to be converted. The *InBuf* parameter points to the first bytes if the truncated sequence. The *OutBuf* and *OutBytesLeft* values are updated according to the number of characters that were previously converted. Because some encoding may have ambiguous data, the **EINVAL** return value has a special meaning at the end of stream conversion. As such, if a user detects an EOF character on a stream that is being converted and the last return code from the **iconv** subroutine was **EINVAL**, the **iconv** subroutine should be called again, with the same *InBytesLeft* parameter and the same character string pointed to by the *InBuf* parameter as when the **EINVAL** return occurred. As a result, the converter will either convert the string as is or declare it an unusable sequence (**EILSEQ**).

Files

Item	Description
/usr/lib/nls/loc/iconv/*	Contains code set converter methods.

iconv_close Subroutine

Purpose

Closes a specified code set converter.

Library

iconv Library (**libiconv.a**)

Syntax

```
#include <iconv.h>
```

```
int iconv_close ( CD)
iconv_t CD;
```

Description

The **iconv_close** subroutine closes a specified code set converter and deallocates any resources used by the converter.

Parameters

Item	Description
CD	Specifies the conversion descriptor to be closed.

Return Values

When successful, the **iconv_close** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **errno** global variable to indicate the error.

Error Codes

The following error code is defined for the **iconv_close** subroutine:

Item	Description
EBADF	The conversion descriptor is not valid.

iconv_open Subroutine

Purpose

Opens a character code set converter.

Library

iconv Library (**libiconv.a**)

Syntax

```
#include <iconv.h>
```

```
iconv_t iconv_open ( ToCode,  FromCode)
const char *ToCode, *FromCode;
```

Description

The **iconv_open** subroutine initializes a code set converter. The code set converter is used by the **iconv** subroutine to convert characters from one code set to another. The **iconv_open** subroutine finds the converter that performs the character code set conversion specified by the *FromCode* and *ToCode* parameters, initializes that converter, and returns a conversion descriptor of type **iconv_t** to identify the code set converter.

The **iconv_open** subroutine first searches the **LOCPATH** environment variable for a converter, using the two user-provided code set names, based on the file name convention that follows:

```
FromCode: "IBM-850"
ToCode: "ISO8859-1"
conversion file: "IBM-850_ISO8859-1"
```

The conversion file name is formed by concatenating the *ToCode* code set name onto the *FromCode* code set name, with an **_** (underscore) between them.

The **LOCPATH** environment variable contains a list of colon-separated directory names. The system default for the **LOCPATH** environment variable is:

```
LOCPATH=/usr/lib/nls/loc
```

See Locales in *Globalization Guide and Reference* for more information on the **LOCPATH** environment variable.

The **iconv_open** subroutine first attempts to find the specified converter in an **iconv** subdirectory under any of the directories specified by the **LOCPATH** environment variable, for example, **/usr/lib/nls/loc/iconv**. If the **iconv_open** subroutine cannot find a converter in any of these directories, it looks for a conversion table in an **iconvTable** subdirectory under any of the directories specified by the **LOCPATH** environment variable, for example, **/usr/lib/nls/loc/iconvTable**.

If the **iconv_open** subroutine cannot find the specified converter in either of these locations, it returns (**iconv_t**) -1 to the calling process and sets the **errno** global variable.

The **iconvTable** directories are expected to contain conversion tables that are the output of the **genxlt** command. The conversion tables are limited to single-byte stateless code sets.

If the named converter is found, the **iconv_open** subroutine will perform the **load** subroutine operation and initialize the converter. A converter descriptor (**iconv_t**) is returned.

Note: When a process calls the **exec** subroutine or a **fork** subroutine, all of the opened converters are discarded.

The **iconv_open** subroutine links the converter function using the **load** subroutine, which is similar to the **exec** subroutine and effectively performs a run-time linking of the converter program. Since the **iconv_open** subroutine is called as a library function, it must ensure that security is preserved for certain programs. Thus, when the **iconv_open** subroutine is called from a set root ID program (a program with permission **—-s—s—x**), it will ignore the **LOCPATH** environment variable and search for converters only in the **/usr/lib/nls/loc/iconv** directory.

Parameters

Item	Description
ToCode	Specifies the destination code set.
FromCode	Specifies the originating code set.

Return Values

A conversion descriptor (**iconv_t**) is returned if successful. Otherwise, the subroutine returns -1, and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EINVAL	The conversion specified by the *FromCode* and *ToCode* parameters is not supported by the implementation.
EMFILE	The number of file descriptors specified by the **OPEN_MAX** configuration variable is currently open in the calling process.

Item	Description
ENFILE	Too many files are currently open in the system.
ENOMEM	Insufficient storage space is available.

Files

Item	Description
/usr/lib/nls/loc/iconv	Contains loadable method converters.
/usr/lib/nls/loc/iconvTable	Contains conversion tables for single-byte stateless code sets.

idlok Subroutine

Purpose

Allows curses to use the hardware insert/delete line feature.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
idlok( Window,  Flag)
WINDOW *Window;
bool Flag;
```

Description

The **idlok** subroutine enables curses to use the hardware insert/delete line feature for terminals so equipped. If this feature is disabled, curses cannot use it. The insert/delete line feature is always considered. Enable this option only if your application needs the insert/delete line feature; for example, for a screen editor. If the insert/delete line feature cannot be used, curses will redraw the changed portions of all lines that do not match the desired line.

Parameters

Item	Description
Flag	Specifies whether to enable curses to use the hardware insert/delete line feature (True) or not (False).
Window	Specifies the window it will affect.

Examples

1. To enable curses to use the hardware insert/delete line feature in stdscr, enter:

   ```
   idlok(stdscr, TRUE);
   ```

2. To force curses not to use the hardware insert/delete line feature in the user-defined window `my_window` , enter:

```
idlok(my_window, FALSE);
```

ilogbd32, ilogbd64, and ilogbd128 Subroutines

Purpose

Returns an unbiased exponent.

Syntax

```
#include <math.h>

int ilogbd32 (x)
_Decimal32 x;

int ilogbd64 (x)
_Decimal64 x;

int ilogbd128 (x)
_Decimal128 x;
```

Description

The **ilogbd32**, **ilogbd64**, and **ilogbd128** subroutines return the integral part of $\log_r |x|$ as a signed integral value, for nonzero *x*, where *r* is the radix of the machine's floating-point arithmetic (r=10).

An application that wants to check for error situations set the **errno** global variable to zero and call the **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. On return, if the **errno** is of the value of nonzero or the **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is of the value of nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **ilogbd32**, **ilogbd64**, and **ilogbd128** subroutines return the exponent part of *x* as a signed integer value. They are equivalent to calling the corresponding **logb** functions and casting the returned value to type **int**.

If *x* is 0, a domain error occurs, and the value **FP_ILOGB0** is returned.

If *x* is ±Inf, a domain error occurs, and the value **{INT_MAX}** is returned.

If *x* is a NaN, a domain error occurs, and the value **FP_ILOGBNAN** is returned.

If the correct value is greater than **{INT_MAX}**, **{INT_MAX}** is returned and a domain error occurs.

If the correct value is less than **{INT_MIN}**, **{INT_MIN}** is returned and a domain error occurs.

ilogbf, ilogbl, or ilogb Subroutine

Purpose

Returns an unbiased exponent.

Syntax

```
#include <math.h>

int ilogbf (x)
float x;

int ilogbl (x)
long double x;

int ilogb (x)
double x;
```

Description

The **ilogbf**, **ilogbl**, and **ilogb** subroutines return the exponent part of the *x* parameter. The return value is the integral part of $\log_r |x|$ as a signed integral value, for nonzero *x*, where *r* is the radix of the machine's floating-point arithmetic (r=2).

An application wishing to check for error situations should set thre **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **ilogbf**, **ilogbl**, and **ilogb** subroutines return the exponent part of *x* as a signed integer value. They are equivalent to calling the corresponding **logb** function and casting the returned value to type **int**.

If *x* is 0, a domain error occurs, and the value **FP_ILOGB0** is returned.

If *x* is ±Inf, a domain error occurs, and the value **{INT_MAX}** is returned.

If *x* is a NaN, a domain error occurs, and the value **FP_ILOGBNAN** is returned.

If the correct value is greater than **{INT_MAX}**, **{INT_MAX}** is returned and a domain error occurs.

If the correct value is less than **{INT_MIN}**, **{INT_MIN}** is returned and a domain error occurs.

imaxabs Subroutine

Purpose

Returns absolute value.

Syntax

```
#include <inttypes.h>

intmax_t imaxabs (j)
intmax_t j;
```

Description

The **imaxabs** subroutine computes the absolute value of an integer *j*. If the result cannot be represented, the behavior is undefined.

Parameters

Item	Description
j	Specifies the value to be computed.

Return Values

The **imaxabs** subroutine returns the absolute value.

imaxdiv Subroutine

Purpose

Returns quotient and remainder.

Syntax

```
#include <inttypes.h>

imaxdiv_t imaxdiv (numer, denom)
intmax_t numer;
intmax_t denom;
```

Description

The **imaxdiv** subroutine computes *numer / denom* and *numer % denom* in a single operation.

Parameters

Item	Description
numer	Specifies the numerator value to be computed.
denom	Specifies the denominator value to be computed.

Return Values

The **imaxdiv** subroutine returns a structure of type **imaxdiv_t**, comprising both the quotient and the remainder. The structure contains (in either order) the members *quot* (the quotient) and *rem* (the remainder), each of which has type **intmax_t**.

If either part of the result cannot be represented, the behavior is undefined.

IMAIXMapping Subroutine

Purpose

Translates a pair of *Key* and *State* parameters to a string and returns a pointer to this string.

Library

Input Method Library (**libIM.a**)

Syntax

```
caddr_t IMAIXMapping(IMMap, Key, State, NBytes)
IMMap  IMMap;
```

```
KeySym  Key;
uint  State;
int * NBytes;
```

Description

The **IMAIXMapping** subroutine translates a pair of *Key* and *State* parameters to a string and returns a pointer to this string.

This function handles the diacritic character sequence and Alt-NumPad key sequence.

Parameters

Item	Description
IMMap	Identifies the keymap.
Key	Specifies the key symbol to which the string is mapped.
State	Specifies the state to which the string is mapped.
NBytes	Returns the length of the returning string.

Return Values

If the length set by the *NBytes* parameter has a positive value, the **IMAIXMapping** subroutine returns a pointer to the returning string.

Note: The returning string is not null-terminated.

IMAuxCreate Callback Subroutine

Purpose

Tells the application program to create an auxiliary area.

Syntax

```
int IMAuxCreate( IM,  AuxiliaryID,  UData)
IMObject IM;
caddr_t *AuxiliaryID;
caddr_t UData;
```

Description

The **IMAuxCreate** subroutine is invoked by the input method of the operating system to create an auxiliary area. The auxiliary area can contain several different forms of data and is not restricted by the interface.

Most input methods display one auxiliary area at a time, but callbacks must be capable of handling multiple auxiliary areas.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
AuxiliaryID	Identifies the newly created auxiliary area.

Item	Description
UData	Identifies an argument passed by the **IMCreate** subroutine.

Return Values

On successful return of the **IMAuxCreate** subroutine, a newly created auxiliary area is set to the *AuxiliaryID* value and the **IMError** global variable is returned. Otherwise, the **IMNoError** value is returned.

IMAuxDestroy Callback Subroutine

Purpose

Tells the application to destroy the auxiliary area.

Syntax

```
int IMAuxDestroy( IM,  AuxiliaryID,  UData)
IMObject IM;
caddr_t AuxiliaryID;
caddr_t UData;
```

Description

The **IMAuxDestroy** subroutine is called by the input method of the operating system to tell the application to destroy an auxiliary area.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
AuxiliaryID	Identifies the auxiliary area to be destroyed.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMAuxDestroy** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

IMAuxDraw Callback Subroutine

Purpose

Tells the application program to draw the auxiliary area.

Syntax

```
int IMAuxDraw(IM, AuxiliaryID, AuxiliaryInformation, UData)
IMObject  IM;
caddr_t  AuxiliaryID;
IMAuxInfo * AuxiliaryInformation;
caddr_t  UData;
```

Description

The **IMAuxDraw** subroutine is invoked by the input method to draw an auxiliary area. The auxiliary area should have been previously created.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
AuxiliaryID	Identifies the auxiliary area.
AuxiliaryInformation	Points to the **IMAuxInfo** structure.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMAuxDraw** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

IMAuxHide Callback Subroutine

Purpose

Tells the application program to hide an auxiliary area.

Syntax

int IMAuxHide(*IM*, *AuxiliaryID*, *UData***)**

```
IMObject IM;
caddr_t AuxiliaryID;
caddr_t UData;
```

Description

The **IMAuxHide** subroutine is called by the input method to hide an auxiliary area.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
AuxiliaryID	Identifies the auxiliary area to be hidden.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMAuxHide** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

IMBeep Callback Subroutine

Purpose

Tells the application program to emit a beep sound.

Syntax

```
int IMBeep( IM,  Percent,  UData)
IMObject IM;
int Percent;
caddr_t UData;
```

Description

The **IMBeep** subroutine tells the application program to emit a beep sound.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
Percent	Specifies the beep level. The value range is from -100 to 100, inclusively. A -100 value means no beep.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMBeep** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

IMClose Subroutine

Purpose

Closes the input method.

Library

Input Method Library (**libIM.a**)

Syntax

```
void IMClose( IMfep)
IMFep IMfep;
```

Description

The **IMClose** subroutine closes the input method. Before the **IMClose** subroutine is called, all previously created input method instances must be destroyed with the **IMDestroy** subroutine, or memory will not be cleared.

Parameters

Item	Description
IMfep	Specifies the input method.

IMCreate Subroutine

Purpose

Creates one instance of an **IMObject** object for a particular input method.

Library

Input Method Library (**libIM.a**)

Syntax

```
IMObject IMCreate( IMfep,  IMCallback,  UData)
IMFep IMfep;
IMCallback *IMCallback;
caddr_t UData;
```

Description

The **IMCreate** subroutine creates one instance of a particular input method. Several input method instances can be created under one input method.

Parameters

Item	Description
IMfep	Specifies the input method.
IMCallback	Specifies a pointer to the caller-supplied **IMCallback** structure.
UData	Optionally specifies an application's own information to the callback functions. With this information, the application can avoid external references from the callback functions. The input method does not change this parameter, but merely passes it to the callback functions. The *UData* parameter is usually a pointer to the application data structure, which contains the information about location, font ID, and so forth.

Return Values

The **IMCreate** subroutine returns a pointer to the created input method instance of type **IMObject**. If the subroutine is unsuccessful, a null value is returned and the **imerrno** global variable is set to indicate the error.

IMDestroy Subroutine

Purpose

Destroys an input method instance.

Library

Input Method Library (**libIM.a**)

Syntax

```
void IMDestroy( IM)
IMObject IM;
```

Description

The **IMDestroy** subroutine destroys an input method instance.

Parameters

Item	Description
IM	Specifies the input method instance to be destroyed.

IMFilter Subroutine

Purpose

Determines if a keyboard event is used by the input method for internal processing.

Library

Input Method Library (**libIM.a**)

Syntax

```
int IMFilter(Im, Key, State, String, Length)
IMObect  Im;
Keysym  Key;
uint  State, * Length;
caddr_t  * String;
```

Description

The **IMFilter** subroutine is used to process a keyboard event and determine if the input method for this operating system uses this event. The return value indicates:

- The event is filtered (used by the input method) if the return value is **IMInputUsed**. Otherwise, the input method did not accept the event.
- Independent of the return value, a string may be generated by the keyboard event if pre-editing is complete.

  Note: The buffer returned from the **IMFilter** subroutine is owned by the input method editor and can not continue between calls.

Parameters

Item	Description
Im	Specifies the input method instance.
Key	Specifies the keysym for the event.
State	Defines the state of the keysym. A value of 0 means that the keysym is not redefined.
String	Holds the returned string if one exists. A null value means that no composed string is ready.
Length	Defines the length of the input string. If the string is not null, returns the length.

Return Values

Item	Description
IMInputUsed	The input method for this operating system filtered the event.
IMInputNotUsed	The input method for this operating system did not use the event.

IMFreeKeymap Subroutine

Purpose

Frees resources allocated by the **IMInitializeKeymap** subroutine.

Library

Input Method Library (**libIM.a**)

Syntax

```
void IMFreeKeymap( IMMap)
IMMap IMMap;
```

Description

The **IMFreeKeymap** subroutine frees resources allocated by the **IMInitializeKeymap** subroutine.

Parameters

Item	Description
IMMap	Identifies the keymap.

IMIndicatorDraw Callback Subroutine

Purpose

Tells the application program to draw the indicator.

Syntax

```
int IMIndicatorDraw( IM,  IndicatorInformation,  UData)
IMObject IM;
IMIndicatorInfo *IndicatorInformation;
caddr_t UData;
```

Description

The **IMIndicatorDraw** callback subroutine is called by the input method when the value of the indicator is changed. The application program then draws the indicator.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.

Item	Description
IndicatorInformation	Points to the **IMIndicatorInfo** structure that holds the current value of the indicator. The interpretation of this value varies among phonic languages. However, the input method provides a function to interpret this value.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error happens, the **IMIndicatorDraw** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

IMIndicatorHide Callback Subroutine

Purpose

Tells the application program to hide the indicator.

Syntax

```
int IMIndicatorHide( IM,  UData)
IMObject IM;
caddr_t UData;
```

Description

The **IMIndicatorHide** subroutine is called by the input method to tell the application program to hide the indicator.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
UData	Specifies an argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMIndicatorHide** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

IMInitialize Subroutine

Purpose

Initializes the input method for a particular language.

Library

Input Method Library (**libIM.a**)

Syntax

```
IMFep IMInitialize( Name)
char *Name;
```

Description

The **IMInitialize** subroutine initializes an input method. The **IMCreate**, **IMFilter**, and **IMLookupString** subroutines use the input method to perform input processing of keyboard events in the form of keysym state modifiers. The **IMInitialize** subroutine finds the input method that performs the input processing specified by the *Name* parameter and returns an Input Method Front End Processor (**IMFep**) descriptor.

Before calling any of the key event-handling functions, the application must create an instance of an *IMObject* object using the **IMFep** descriptor. Each input method can produce one or more instances of *IMObject* object with the **IMCreate** subroutine.

When the **IMInitialize** subroutine is called, strings returned from the input method are encoded in the code set of the locale. Each **IMFep** description inherits the code set of the locale when the input method is initialized. The locale setting does not change the code set of the **IMFep** description after it is created.

The **IMInitialize** subroutine calls the **load** subroutine to load a file whose name is in the form *Name*.**im**. The *Name* parameter is passed to the **IMInitialize** subroutine. The loadable input method file is accessed in the directories specified by the **LOCPATH** environment variable. The default location for loadable input-method files is the **/usr/lib/nls/loc** directory. If none of the **LOCPATH** directories contain the input method specified by the *Name* parameter, the default location is searched.

Note: All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

The name of the input method file usually corresponds to the locale name, which is in the form **Language_territory.codesest@***modifier*. In the environment, the modifier is in the form **@im**=*modifier*. The **IMInitialize** subroutine converts the **@im**= substring to **@** when searching for loadable input-method files.

Parameters

Item	Description
Name	Specifies the language to be used. Each input method is dynamically linked to the application program.

Return Values

If **IMInitialize** succeeds, it returns an **IMFep** handle. Otherwise, null is returned and the **imerrno** global variable is set to indicate the error.

Files

Item	Description
/usr/lib/nls/loc	Contains loadable input-method files.

IMInitializeKeymap Subroutine

Purpose

Initializes the keymap associated with a specified language.

Library

Input Method Library (**libIM.a**)

Syntax

```
IMMap IMInitializeKeymap( Name)
char *Name;
```

Description

The **IMInitializeKeymap** subroutine initializes an input method keymap (imkeymap). The **IMAIXMapping** and **IMSimpleMapping** subroutines use the imkeymap to perform mapping of keysym state modifiers to strings. The **IMInitializeKeymap** subroutine finds the imkeymap that performs the keysym mapping and returns an imkeymap descriptor, **IMMap**. The strings returned by the imkeymap mapping functions are treated as unsigned bytes.

The applications that use input methods usually do not need to manage imkeymaps separately. The imkeymaps are managed internally by input methods.

The **IMInitializeKeymap** subroutine searches for an imkeymap file whose name is in the form *Name*.**im**. The *Name* parameter is passed to the **IMInitializeKeymap** subroutine. The imkeymap file is accessed in the directories specified by the **LOCPATH** environment variable. The default location for input method files is the **/usr/lib/nls/loc** directory. If none of the **LOCPATH** directories contain the keymap method specified by the *Name* parameter, the default location is searched.

Note: All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

The name of the imkeymap file usually corresponds to the locale name, which is in the form **Language_territory.codesest@***modifier*. In the AIXwindows environment, the modifier is in the form **@im**=*modifier*. The **IMInitializeKeymap** subroutine converts the **@im**= *substring* to **@** (at sign) when searching for loadable input method files.

Parameters

Item	Description
Name	Specifies the name of the imkeymap.

Return Values

The **IMInitializeKeymap** subroutine returns a descriptor of type **IMMap**. Returning a null value indicates the occurrence of an error. The **IMMap** descriptor is defined in the **im.h** file as the **caddr_t** structure. This descriptor is used for keymap manipulation functions.

Files

Item	Description
/usr/lib/nls/loc	Contains loadable input-method files.

IMIoctl Subroutine

Purpose

Performs a variety of control or query operations on the input method.

Library

Input Method Library (**libIM.a**)

Syntax

```
int IMIoctl( IM,  Operation,  Argument)
IMObject IM;
int Operation;
char *Argument;
```

Description

The **IMIoctl** subroutine performs a variety of control or query operations on the input method specified by the *IM* parameter. In addition, this subroutine can be used to control the unique function of each language input method because it provides input method-specific extensions. Each input method defines its own function.

Parameters

IM
Specifies the input method instance.

Operation
Specifies the operation.

Argument
The use of this parameter depends on which of the following operations is performed.

IM_Refresh
Refreshes the text area, auxiliary areas, and indicator by calling the needed callback functions if these areas are not empty. The *Argument* parameter is not used.

IM_GetString
Gets the current pre-editing string. The *Argument* parameter specifies the address of the **IMSTR** structure supplied by the caller. The callback function is invoked to clear the pre-editing if it exists.

IM_Clear
Clears the text and auxiliary areas if they exist. If the *Argument* parameter is not a null value, this operation invokes the callback functions to clear the screen. The keyboard state remains the same.

IM_Reset
Clears the auxiliary area if it currently exists. If the *Argument* parameter is a null value, this operation clears only the internal buffer of the input method. Otherwise, the **IMAuxHide** subroutine is called, and the input method returns to its initial state.

IM_ChangeLength
Changes the maximum length of the pre-editing string.

IM_ChangeMode
Sets the Processing Mode of the input method to the mode specified by the *Argument* parameter. The valid value for *Argument* is:

IMNormalMode
Specifies the normal mode of pre-editing.

IMSuppressedMode
Suppresses pre-editing.

IM_QueryState
Returns the status of the text area, the auxiliary area, and the indicator. It also returns the beep status and the processing mode. The results are stored into the caller-supplied **IMQueryState** structure pointed to by the *Argument* parameter.

IM_QueryText
Returns detailed information about the text area. The results are stored in the caller-supplied **IMQueryText** structure pointed to by the *Argument* parameter.

IM_QueryAuxiliary
Returns detailed information about the auxiliary area. The results are stored in the caller-supplied **IMQueryAuxiliary** structure pointed to by the *Argument* parameter.

IM_QueryIndicator
Returns detailed information about the indicator. The results are stored in the caller-supplied **IMQueryIndicator** structure pointed to by the *Argument* parameter.

IM_QueryIndicatorString
Returns an indicator string corresponding to the current indicator. Results are stored in the caller-supplied **IMQueryIndicatorString** structure pointed to by the *Argument* parameter. The caller can request either a short or long form with the format member of the **IMQueryIndicatorString** structure.

IM_SupportSelection
Informs the input method whether or not an application supports an auxiliary area selection list. The application must support selections inside the auxiliary area and determine how selections are displayed. If this operation is not performed, the input method assumes the application does not support an auxiliary area selection list.

Return Values

The **IMIoctl** subroutine returns a value to the **IMError** global variable that indicates the type of error encountered. Some error types are provided in the **/usr/include/imerrno.h** file.

IMLookupString Subroutine

Purpose

Maps a *Key/State* (key symbol/state) pair to a string.

Library

Input Method Library (**libIM.a**)

Syntax

```
int IMLookupString(Im, Key, State, String, Length)
IMObject  Im;
KeySym  Key;
uint  State, * Length;
caddr_t * String;
```

Description

The **IMLookupString** subroutine is used to map a *Key/State* pair to a localized string. It uses an internal input method keymap (**imkeymap**) file to map a keysym/modifier to a string. The string returned is encoded in the same code set as the locale of **IMObject** and IM Front End Processor.

Note: The buffer returned from the **IMLookupString** subroutine is owned by the input method editor and can not continue between calls.

Parameters

Item	Description
Im	Specifies the input method instance.
Key	Specifies the key symbol for the event.

Item	Description
State	Defines the state for the event. A value of 0 means that the key is not redefined.
String	Holds the returned string, if one exists. A null value means that no composed string is ready.
Length	Defines the length string on input. If the string is not null, identifies the length returned.

Return Values

Item	Description
IMError	Error encountered.
IMReturnNothing	No string or keysym was returned.
IMReturnString	String returned.

IMProcess Subroutine

Purpose

Processes keyboard events and language-specific input.

Library

Input Method Library **(libIM.a**)

Note: This subroutine will be removed in future releases. Use the **IMFilter** and **IMLookupString** subroutines to process keyboard events.

Syntax

```
int IMProcess (IM, KeySymbol, State, String, Length)
IMObject  IM;
KeySym  KeySymbol;
uint  State;
caddr_t * String;
uint * Length;
```

Description

This subroutine is a main entry point to the input method of the operating system. The **IMProcess** subroutine processes one keyboard event at a time. Processing proceeds as follows:

- Validates the *IM* parameter.
- Performs keyboard translation for all supported modifier states.
- Invokes internal function to do language-dependent processing.
- Performs any necessary callback functions depending on the internal state.
- Returns to application, setting the *String* and *Length* parameters appropriately.

Parameters

Item	Description
IM	Specifies the input method instance.
KeySymbol	Defines the set of keyboard symbols that will be handled.
State	Specifies the state of the keyboard.

Item	Description
String	Holds the returned string. Returning a null value means that the input is used or discarded by the input method. **Note:** The *String* parameter is not a null-terminated string.
Length	Stores the length, in bytes, of the *String* parameter.

Return Values

This subroutine returns the **IMError** global variable if an error occurs. The **IMerrno** global variable is set to indicate the error. Some of the variable values include:

Item	Description
IMError	Error occurred during this subroutine.
IMTextAndAuxiliaryOff	No text string in the Text area, and the Auxiliary area is not shown.
IMTextOn	Text string in the Text area, but no Auxiliary area.
IMAuxiliaryOn	No text string in the Text area, and the Auxiliary area is shown.
IMTextAndAuxiliaryOn	Text string in the Text area, and the Auxiliary is shown.

IMProcessAuxiliary Subroutine

Purpose

Notifies the input method of input for an auxiliary area.

Library

Input Method Library (**libIM.a**)

Syntax

```
int IMProcessAuxiliary(IM, AuxiliaryID, Button, PanelRow
      PanelColumn, ItemRow, ItemColumn, String, Length)

IMObject  IM;
caddr_t  AuxiliaryID;
uint  Button;
uint  PanelRow;
uint  PanelColumn;
uint  ItemRow;
uint  ItemColumn;
caddr_t *String;
uint *Length;
```

Description

The **IMProcessAuxiliary** subroutine notifies the input method instance of input for an auxiliary area.

Parameters

Item	Description
IM	Specifies the input method instance.

Item	Description
AuxiliaryID	Identifies the auxiliary area.
Button	Specifies one of the following types of input: **IM_ABORT** Abort button is pushed. **IM_CANCEL** Cancel button is pushed. **IM_ENTER** Enter button is pushed. **IM_HELP** Help button is pushed. **IM_IGNORE** Ignore button is pushed. **IM_NO** No button is pushed. **IM_OK** OK button is pushed. **IM_RETRY** Retry button is pushed. **IM_SELECTED** Selection has been made. Only in this case do the *PanelRow*, *PanelColumn*, *ItemRow*, and *ItemColumn* parameters have meaningful values. **IM_YES** Yes button is pushed.
PanelRow	Indicates the panel on which the selection event occurred.
PanelColumn	Indicates the panel on which the selection event occurred.
ItemRow	Indicates the selected item.
ItemColumn	Indicates the selected item.
String	Holds the returned string. If a null value is returned, the input is used or discarded by the input method. Note that the *String* parameter is not a null-terminated string.
Length	Stores the length, in bytes, of the *String* parameter.

IMQueryLanguage Subroutine

Purpose

Checks to see if the specified input method is supported.

Library

Input Method Library (**libIM.a**)

Syntax

```
uint IMQueryLanguage( Name)
IMLanguage Name;
```

Description

The **IMQueryLanguage** subroutine checks to see if the input method specified by the *Name* parameter is supported.

Parameters

Item	Description
Name	Specifies the input method.

Return Values

The **IMQueryLanguage** subroutine returns a true value if the specified input method is supported, a false value if not.

IMSimpleMapping Subroutine

Purpose

Translates a pair of *KeySymbol* and *State* parameters to a string and returns a pointer to this string.

Library

Input Method Library (**libIM.a**)

Syntax

```
caddr_t IMSimpleMapping (IMMap, KeySymbol, State, NBytes)
IMMap  IMMap;
KeySym  KeySymbol;
uint  State;
int * NBytes;
```

Description

Like the **IMAIXMapping** subroutine, the **IMSimpleMapping** subroutine translates a pair of *KeySymbol* and *State* parameters to a string and returns a pointer to this string. The parameters have the same meaning as those in the **IMAIXMapping** subroutine.

The **IMSimpleMapping** subroutine differs from the **IMAIXMapping** subroutine in that it does not support the diacritic character sequence or the Alt-NumPad key sequence.

Parameters

Item	Description
IMMap	Identifies the keymap.
KeySymbol	Key symbol to which the string is mapped.
State	Specifies the state to which the string is mapped.
NBytes	Returns the length of the returning string.

IMTextCursor Callback Subroutine

Purpose

Asks the application to move the text cursor.

Syntax

```
int IMTextCursor(IM, Direction, Cursor, UData)
IMObject  IM;
uint  Direction;
int * Cursor;
caddr_t  UData;
```

Description

The **IMTextCursor** subroutine is called by the Input Method when the Cursor Up or Cursor Down key is input to the **IMFilter** and **IMLookupString** subroutines.

This subroutine sets the new display cursor position in the text area to the integer pointed to by the *Cursor* parameter. The cursor position is relative to the top of the text area. A value of -1 indicates the cursor should not be moved.

Because the input method does not know the actual length of the screen it always treats a text string as one-dimensional (a single line). However, in the terminal emulator, the text string sometimes wraps to the next line. The **IMTextCursor** subroutine performs this conversion from single-line to multiline text strings. When you move the cursor up or down, the subroutine interprets the cursor position on the text string relative to the input method.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the Input Method instance.
Direction	Specifies up or down.
Cursor	Specifies the new cursor position or -1.
UData	Specifies an argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMTextCursor** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

IMTextDraw Callback Subroutine

Purpose

Tells the application program to draw the text string.

Syntax

```
int IMTextDraw( IM,  TextInfo,  UData)
IMObject IM;
```

```
IMTextInfo *TextInfo;
caddr_t UData;
```

Description

The **IMTextDraw** subroutine is invoked by the Input Method whenever it needs to update the screen with its internal string. This subroutine tells the application program to draw the text string.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
TextInfo	Points to the **IMTextInfo** structure.
UData	An argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMTextDraw** subroutine returns the **IMError** global variable. Otherwise, the **IMNoError** value is returned.

IMTextHide Callback Subroutine

Purpose

Tells the application program to hide the text area.

Syntax

```
int IMTextHide( IM,  UData)
IMObject IM;
caddr_t UData;
```

Description

The **IMTextHide** subroutine is called by the input method when the text area should be cleared. This subroutine tells the application program to hide the text area.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
UData	Specifies an argument passed by the **IMCreate** subroutine.

Return Values

If an error occurs, the **IMTextHide** subroutine returns an **IMError** value. Otherwise, an **IMNoError** value is returned.

IMTextStart Callback Subroutine

Purpose

Notifies the application program of the length of the pre-editing space.

Syntax

```
int IMTextStart( IM,  Space,  UData)
IMObject IM;
int *Space;
caddr_t UData;
```

Description

The **IMTextStart** subroutine is called by the input method when the pre-editing is started, but prior to calling the **IMTextDraw** callback subroutine. This subroutine notifies the input method of the length, in terms of bytes, of pre-editing space. It sets the length of the available space (>=0) on the display to the integer pointed to by the *Space* parameter. A value of -1 indicates that the pre-editing space is dynamic and has no limit.

This subroutine is provided by applications that use input methods.

Parameters

Item	Description
IM	Indicates the input method instance.
Space	Maximum length of pre-editing string.
UData	An argument passed by the **IMCreate** subroutine.

inch, mvinch, mvwinch, or winch Subroutine

Purpose

Inputs a single-byte character and rendition from a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
chtype inch(void);
```

```
chtype mvinch(int y,
int x);
```

```
chtype mvwinch(WINDOW  *win,
int y,
int x);
```

```
chtype winch(WINDOW *win);
```

Description

The **inch**, **winch**, **mvinch**, and **mvwinch** subroutines return the character and rendition, of type chtype, at the current or specified position in the current or specified window.

Parameters

Item	Description
**win*	Specifies the window from which to get the character.
x	
y	

Return Values

Upon successful completion, these subroutines return the specified character and rendition. Otherwise, they return (chtype) ERR.

Examples

1. To get the character at the current cursor location in the stdscr, enter:

```
chtype character;

character = inch();
```

2. To get the character at the current cursor location in the user-defined window `my_window`, enter:

```
WINDOW *my_window;
chtype character;

character = winch(my_window);
```

3. To move the cursor to the coordinates y = `0`, x = 5 and then get that character, enter:

```
chtype character;

character = mvinch(0, 5);
```

4. To move the cursor to the coordinates y = `0`, x = 5 in the user-defined window `my_window` and then get that character, enter:

```
WINDOW *my_window;
chtype character;

character = mvwinch(my_window, 0, 5);
```

inet_aton Subroutine

Purpose

Converts an ASCII string into an Internet address.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

int inet_aton (*CharString*, *InternetAddr***)**
char * *CharString*;
struct in_addr * *InternetAddr*;

Description

The **inet_aton** subroutine takes an ASCII string representing the Internet address in dot notation and converts it into an Internet address.

All applications containing the **inet_aton** subroutine must be compiled with **_BSD** set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Parameters

Item	Description
CharString	Contains the ASCII string to be converted to an Internet address.
InternetAddr	Contains the Internet address that was converted from the ASCII string.

Return Values

Upon successful completion, the **inet_aton** subroutine returns 1 if *CharString* is a valid ASCII representation of an Internet address.

The **inet_aton** subroutine returns 0 if *CharString* is not a valid ASCII representation of an Internet address.

Files

Item	Description
/etc/hosts	Contains host names.
/etc/networks	Contains network names.

init_color Subroutine

Purpose

Changes a color definition.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

init_color( Color, R,
G, B)
register short Color,  R,  G,  B;
```

Description

The **init_color** subroutine changes a color definition. A single color is defined by the combination of its red, green, and blue components. The **init_color** subroutine changes all the occurrences of the color on the screen immediately. If the color is changed successfully, this subroutines returns OK. Otherwise, it returns ERR.

Note: The values for the red, green, and blue components must be between 0 (no component) and 1000 (maximum amount of component). The **init_color** subroutine sets values less than 0 to 0 and values greater than 1000 to 1000.

To determine if you can change a terminal's color definitions, see the **can_change_color** subroutine.

Return Values

Item	Description
OK	Indicates the color was changed successfully.
ERR	Indicates the color was not changed.

Parameters

Item	Description
Color	Identifies the color to change. The value of the parameter must be between **0** and **COLORS-1**.
R	Specifies the desired intensity of the red component.
G	Specifies the desired intensity of the green component.
B	Specifies the desired intensity of the blue component.

Examples

To initialize the color definition for color 11 to violet on a terminal that supports at least 12 colors, use:

```
init_color(11,500,0,500);
```

init_pair Subroutine

Purpose

Changes a color-pair definition.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

init_pair( Pair, F, B)
register short Pair,  F,  B;
```

Description

The **init_pair** subroutine changes a color-pair definition. A color pair is a combination of a foreground and a background color. If you specify a color pair that was previously initialized, curses refreshes the screen and changes all occurrences of that color pair to the new definition. You must call the **start_color** subroutine before you call this subroutine.

Return Values

Item	Description
OK	Indicates successful completion.
ERR	Indicates the subroutine failed.

Parameters

Item	Description
Pair	Identifies the color-pair number. The value of the *Pair* parameter must be between **1** and **COLORS_PAIRS-1**.
F	Specifies the foreground color number. This number must be between **0** and **COLORS-1**.
B	Specifies the background color number. This number must be between **0** and **COLORS-1**.

Examples

To initialize the color definition for color-pair 2 to a black foreground (color 0) with a cyan background (color 3), use:

```
init_pair(2,COLOR_BLACK, COLOR_CYAN);
```

initgroups Subroutine

Purpose

Initializes supplementary group ID.

Library

Standard C Library (**libc.a**)

Syntax

```
int initgroups ( User,  BaseGID)
const char *User;
int BaseGID;
```

Description

Attention: The **initgroups** subroutine uses the **getgrent** and **getpwent** family of subroutines. If the program that invokes the **initgroups** subroutine uses any of these subroutines, calling the **initgroups** subroutine overwrites the static storage areas used by these subroutines.

The **initgroups** subroutine reads the defined group membership of the specified *User* parameter and sets the supplementary group ID of the current process to that value. The *BaseGID* parameter is always

included in the supplementary group ID. The supplementary group is normally the principal user's group. If the user is in more than **NGROUPS_MAX** groups, set in the **limits.h** file, only **NGROUPS_MAX** groups are set, including the *BaseGID* group.

Parameters

Item	Description
User	Identifies a user.
BaseGID	Specifies an additional group to include in the group set.

Return Values

Item	Description
0	Indicates that the subroutine was success.
-1	Indicates that the subroutine failed. The **errno** global variable is set to indicate the error.

initialize Subroutine

Purpose

Performs printer initialization.

Library

None (provided by the formatter).

Syntax

```
#include <piostruct.h>

int initialize ()
```

Description

The **initialize** subroutine is invoked by the formatter driver after the **setup**subroutine returns.

If the **-j** flag passed from the **qprt** command has a nonzero value (true), the **initialize** subroutine uses the **piocmdout** subroutine to send a command string to the printer. This action initializes the printer to the proper state for printing the file. Any variables referenced by the command string should be the attribute values from the database, overridden by values from the command line.

If the **-j** flag passed from the **qprt** command has a nonzero value (true), any necessary fonts should be downloaded.

Return Values

Item	Description
0	Indicates a successful operation.

If the **initialize** subroutine detects an error, it uses the **piomsgout** subroutine to invoke an error message. It then invokes the **pioexit** subroutine with a value of **PIOEXITBAD**.

Note: If either the **piocmdout** or **piogetstr** subroutine detects an error, it issues its own error messages and terminates the print job.

initlabeldb and endlabeldb Subroutines

Purpose

Initializes or terminates database.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>
int initlabeldb (dbfile)
const char * dbfile;

int endlabeldb (void)
```

Description

The **initlabeldb** subroutine initializes the label database that the *dbfile* parameter specifies. When the *dbfile* parameter is specified to NULL, the **initlabeldb** subroutine initializes the library data members using the **/etc/security/enc/LabelEncodings** file. The **initlabeldb** subroutine succeeds only if the formation of the label file is correct.

Before any operations on a label, must use the **initlabeldb** subroutine to initialize the database. The database that is initialized will be read only.

The **endlabeldb** subroutine terminates the database by freeing all of the memory that is allocated. There is no write back in this operation.

Parameters

Item	Description
dbfile	Specifies the file name that is to be used for label database initialization.

Security

Access Control: To access the default encodings file **/etc/security/enc/LabelEncodings**, the process must have the **PV_LAB_LEF** privilege.

File Accessed

Mode	File
r	**/etc/security/enc/LabelEncodings**

Return Values

If successful, the **initlabeldb** and **endlabeldb** subroutines return a value of zero. Otherwise, they return a value of -1.

Errors

If the **initlabeldb** subroutine fails, one of the following **errno** values can be set:

Item	Description
EBADF	The parameter that is passed is not NULL and is not a regular file.
EALREADY	The database specified is already initialized with a different encoding file.
EACCESS	The operation is not permitted.
ENOENT	The label encoding file is not found.

If the **endlabeldb** subroutine fails, it returns the following **errno** value:

Item	Description
ENOTREADY	The database is not initialized.

insch, mvinsch, mvwinsch, or winsch Subroutine

Purpose

Inserts a single-byte character and rendition in a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int insch(chtype ch);
```

```
int mvinsch(int y,
chtype h);
```

```
int mvwinsch(WINDOW  *win,
int x,
int y,
chtype h);
```

```
int winsch(WINDOW *win,
chtype h);
```

Description

These subroutines insert the character and rendition into the current or specified window at the current or specified position.

These subroutines do not perform wrapping or advance the cursor position. These functions perform special-character processing, with the exception that if a **newline** is inserted into the last line of a window and scrolling is not enabled, the behavior is unspecified.

Parameters

Item	Description
ch	
y	

Item	Description
x	
**win*	Specifies the window in which to insert the character.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To insert the character x in the stdscr, enter:

```
chtype x;
insch(x);
```

2. To insert the character x into the user-defined window `my_window`, enter:

```
WINDOW *my_window
chtype x;
winsch(my_window, x);
```

3. To move the logical cursor to the coordinates Y=10, X=5 prior to inserting the character x in the stdscr, enter:

```
chtype x;
mvinsch(10, 5, x);
```

4. To move the logical cursor to the coordinates y=10, X=5 prior to inserting the character x in the user-defined window `my_window`, enter:

```
WINDOW *my_window;
chtype x;
mvwinsch(my_window, 10, 5, x);
```

insertln or winsertln Subroutine

Purpose

Inserts a blank line above the current line in a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int insertln(void)
```

int winsertln(WINDOW **win*);

Description

The **insertln** and **winsertln** subroutines insert a blank line before the current line in the current or specified window. The bottom line is no longer displayed. The cursor position does not change.

Parameters

Item	Description
**win*	Specifies the window in which to insert the blank line.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To insert a blank line above the current line in the stdscr, enter:

```
insertln();
```

2. To insert a blank line above the current line in the user-defined window `my_window`, enter:

```
WINDOW *mywindow;
winsertln(my_window);
```

insque or remque Subroutine

Purpose

Inserts or removes an element in a queue.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <search.h>
```

insque (*Element*, *Pred*)
void ***Element*, ***Pred**;

```
remque (Element)
void *Element;
```

Description

The **insque** and **remque** subroutines manipulate queues built from double-linked lists. Each element in the queue must be in the form of a **qelem** structure. The **next** and **prev** elements of that structure must point to the elements in the queue immediately before and after the element to be inserted or deleted.

The **insque** subroutine inserts the element pointed to by the *Element* parameter into a queue immediately after the element pointed to by the *Pred* parameter.

The **remque** subroutine removes the element defined by the *Element* parameter from a queue.

Parameters

Item	Description
Pred	Points to the element in the queue immediately before the element to be inserted or deleted.

Item	Description
Element	Points to the element in the queue immediately after the element to be inserted or deleted.

install_lwcf_handler Subroutine

Purpose

Registers the signal handler to dump a lightweight core file for signals that normally cause the generation of a core file.

Library

PTools Library (**libptools_ptr.a**)

Syntax

```
void install_lwcf_handler (void);
```

Description

The **install_lwcf_handler** subroutine registers the signal handler to dump a lightweight core file for signals that normally cause a core file to be generated. The format of lightweight core files complies with the Parallel Tools Consortium Lightweight Core File Format.

The **install_lwcf_handler** subroutine uses the **LIGHTWEIGHT_CORE** environment variable to determine the target lightweight core file. If the **LIGHTWEIGHT_CORE** environment variable is defined, a lightweight core file will be generated. Otherwise, a normal core file will be generated.

If the **LIGHTWEIGHT_CORE** environment variable is defined without a value, the lightweight core file is assigned the default file name **lw_core** and is created under the current working directory if it does not already exist.

If the **LIGHTWEIGHT_CORE** environment variable is defined with a value of **STDERR**, the lightweight core file is output to the standard error output device of the process. Keyword STDERR is not case-sensitive.

If the **LIGHTWEIGHT_CORE** environment variable is defined with the value of a character string other than *STDERR*, the string is used as a path name for the lightweight core file generated.

If the target lightweight core file already exists, the traceback information is appended to the file.

The **install_lwcf_handler** subroutine can be called directly from an application to register the signal handler. Alternatively, linker option **-binitfini:install_lwcf_handler** can be used when linking an application, which specifies to execute the **install_lwcf_handler** subroutine when the application is initialized. The advantage of the second method is that the application code does not need to change to invoke the **install_lwcf_handler** subroutine.

Note: The source line information in a `Lightweight_core` file is not displayed by default when the text page size is 64 K. When the text page size is 64K, use the environment variable `AIX_LDSYM=ON` to get the source line information in a `Lightweight_core` file.

intrflush Subroutine

Purpose

Enables or disables flush on interrupt.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

int intrflush(WINDOW * *win*,
bool *bf*);

Description

The **intrflush** subroutine specifies whether pressing an interrupt key (interrupt, suspend, or quit) will flush the input buffer associated with the current screen. If the value of *bf* is TRUE, then flushing of the output buffer associated with the current screen will occur when an interrupt key (interrupt, suspend, or quit) is pressed. If the value of *bf* is FALSE then no flushing of the buffer will occur when an interrupt key is pressed. The default for the option is inherited from the display driver settings. The *win* argument is ignored.

Parameters

Item	Description
bf	
**win*	Specifies the window for which to enable or disable queue flushing.

Return Values

Upon successful completion, the **intrflush** subroutine returns OK. Otherwise, it returns ERR.

Examples

1. To enable queue flushing in the user-defined window `my_window`, enter:

   ```
   intrflush(my_window, TRUE);
   ```

2. To disable queue flushing in the user-defined window `my_window`, enter:

   ```
   intrflush(my_window, FALSE);
   ```

ioctl, ioctlx, ioctl32, or ioctl32x Subroutine

Purpose

Performs control functions associated with open file descriptors.

Library

Standard C Library (**libc.a**)

BSD Library (**libbsd.a**)

Syntax

#include <sys/ioctl.h> #include <sys/types.h> #include <unistd.h> #include <stropts.h>

int ioctl (*FileDescriptor***,** *Command***,** *Argument***) int** *FileDescriptor*, *Command***; void *** *Argument***;**

int ioctlx (*FileDescriptor***,** *Command***,** *Argument***,** *Ext* **) int** *FileDescriptor* **,** *Command* **; void ****Argument***;**
int *Ext***;**

int ioct132 (*FileDescriptor***,** *Command* **,** *Argument***) int** *FileDescriptor***,** *Command***; unsigned int** *Argument***;**

int ioct132x (*FileDescriptor***, *Command* , *Argument*, *Ext*) int** *FileDescriptor*, *Command*; **unsigned int** *Argument*; **unsigned int** *Ext*;

Description

The **ioctl** subroutine performs a variety of control operations on the object associated with the specified open file descriptor. This function is typically used with character or block special files, **sockets**, or generic device support such as the **termio** general terminal interface.

The control operation provided by this function call is specific to the object being addressed, as are the data type and contents of the *Argument* parameter. The **ioctlx** form of this function can be used to pass an additional extension parameter to objects supporting it. The **ioct132** and **ioct132x** forms of this function behave in the same way as **ioctl** and **ioctlx**, but allow 64-bit applications to call the **ioctl** routine for an object that does not normally work with 64-bit applications.

Performing an ioctl function on a file descriptor associated with an ordinary file results in an error being returned.

Parameters

Item	Description
FileDescriptor	Specifies the open file descriptor for which the control operation is to be performed.
Command	Specifies the control function to be performed. The value of this parameter depends on which object is specified by the *FileDescriptor* parameter.
Argument	Specifies additional information required by the function requested in the *Command* parameter. The data type of this parameter (a **void** pointer) is object-specific, and is typically used to point to an object device-specific data structure. However, in some device-specific instances, this parameter is used as an integer.
Ext	Specifies an extension parameter used with the **ioctlx** subroutine. This parameter is passed on to the object associated with the specified open file descriptor. Although normally of type **int**, this parameter can be used as a pointer to a device-specific structure for some devices.

File Input/Output (FIO) ioctl Command Values

A number of file input/output (FIO) ioctl commands are available to enable the **ioctl** subroutine to function similar to the **fcntl** subroutine:

Item	Description

FIOCLEX and FIONCLEX	Manipulate the **close-on-exec** flag to determine if a file descriptor should be closed as part of the normal processing of the **exec** subroutine. If the flag is set, the file descriptor is closed. If the flag is clear, the file descriptor is left open.
	
	The following code sample illustrates the use of the **fcntl** subroutine to set and clear the **close-on-exec** flag:
	
	`/* set the close-on-exec flag for fd1 */`
	`fcntl(fd1,F_SETFD,FD_CLOEXEC);`
	`/* clear the close-on-exec flag for fd2 */`
	`fcntl(fd2,F_SETFD,0);`
	
	Although the **fcntl** subroutine is normally used to set the **close-on-exec** flag, the **ioctl** subroutine may be used if the application program is linked with the Berkeley Compatibility Library (**libbsd.a**) or the Berkeley Thread Safe Library (**libbsd_r.a**). The following ioctl code fragment is equivalent to the preceding **fcntl** fragment:
	
	`/* set the close-on-exec flag for fd1 */`
	`ioctl(fd1,FIOCLEX,0);`
	`/* clear the close-on-exec flag for fd2 */`
	`ioctl(fd2,FIONCLEX,0);`
	
	The third parameter to the **ioctl** subroutine is not used for the **FIOCLEX** and **FIONCLEX** ioctl commands.

FIONBIO	Enables nonblocking I/O. The effect is similar to setting the **O_NONBLOCK** flag with the **fcntl** subroutine. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer that indicates whether nonblocking I/O is being enabled or disabled. A value of 0 disables non-blocking I/O. Any nonzero value enables nonblocking I/O. A sample code fragment follows:
	
	`int flag;`
	`/* enable NBIO for fd1 */`
	`flag = 1;`
	`ioctl(fd1,FIONBIO,&flag);`
	`/* disable NBIO for fd2 */`
	`flag = 0;`
	`ioctl(fd2,FIONBIO,&flag);`

FIONREAD	Determines the number of bytes that are immediately available to be read on a file descriptor. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer variable where the byte count is to be returned. The following sample code illustrates the proper use of the **FIONREAD** ioctl command:
	
	`int nbytes;`
	
	`ioctl(fd,FIONREAD,&nbytes);`

Item	Description

FIOASYNC	Enables a simple form of asynchronous I/O notification. This command causes the kernel to send **SIGIO** signal to a process or a process group when I/O is possible. Only sockets, ttys, and pseudo-ttys implement this functionality.
	
	The third parameter of the **ioctl** subroutine for this command is a pointer to an integer variable that indicates whether the asynchronous I/O notification should be enabled or disabled. A value of 0 disables I/O notification; any nonzero value enables I/O notification. A sample code segment follows:
	
	`int flag;`
	`/* enable ASYNC on fd1 */`
	`flag = 1;`
	`ioctl(fd, FIOASYNC,&flag);`
	`/* disable ASYNC on fd2 */`
	`flag = 0;`
	`ioctl(fd,FIOASYNC,&flag);`

FIOSETOWN	Sets the recipient of the **SIGIO** signals when asynchronous I/O notification (**FIOASYNC**) is enabled. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer that contains the recipient identifier. If the value of the integer pointed to by the third parameter is negative, the value is assumed to be a process group identifier. If the value is positive, it is assumed to be a process identifier.
	
	Sockets support both process groups and individual process recipients, while ttys and psuedo-ttys support only process groups. Attempts to specify an individual process as the recipient will be converted to the process group to which the process belongs. The following code example illustrates how to set the recipient identifier:
	
	`int owner;`
	`owner = -getpgrp();`
	`ioctl(fd,FIOSETOWN,&owner);`
	
	Note: In this example, the asynchronous I/O signals are being enabled on a process group basis. Therefore, the value passed through the owner parameter must be a negative number.
	
	The following code sample illustrates enabling asynchronous I/O signals to an individual process:
	
	`int owner;`
	`owner = getpid();`
	`ioctl(fd,FIOSETOWN,&owner);`

FIOGETOWN	Determines the current recipient of the asynchronous I/O signals of an object that has asynchronous I/O notification (**FIOASYNC**) enabled. The third parameter to the **ioctl** subroutine for this command is a pointer to an integer used to return the owner ID. For example:
	
	`int owner;`
	`ioctl(fd,FIOGETOWN,&owner);`
	
	If the owner of the asynchronous I/O capability is a process group, the value returned in the reference parameter is negative. If the owner is an individual process, the value is positive.

Return Values

If the **ioctl** subroutine fails, a value of -1 is returned. The **errno** global variable is set to indicate the error.

The **ioctl** subroutine fails if one or more of the following are true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid open file descriptor.
EFAULT	The *Argument* or *Ext* parameter is used to point to data outside of the process address space.
EINTR	A signal was caught during the **ioctl** or **ioctlx** subroutine and the process had not enabled re-startable subroutines for the signal.
EINTR	A signal was caught during the **ioctl** , **ioctlx** , **ioctl32** , or **ioct132x** subroutine and the process had not enabled re-startable subroutines for the signal.
EINVAL	The *Command* or *Argument* parameter is not valid for the specified object.
ENOTTY	The *FileDescriptor* parameter is not associated with an object that accepts control functions.
ENODEV	The *FileDescriptor* parameter is associated with a valid character or block special file, but the supporting device driver does not support the ioctl function.
ENXIO	The *FileDescriptor* parameter is associated with a valid character or block special file, but the supporting device driver is not in the configured state.

Object-specific error codes are defined in the documentation for associated objects.

is_linetouched, is_wintouched, touchline, touchwin, untouchwin, or wtouchin Subroutine

Purpose

Window refresh control functions.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

bool is_linetouched(WINDOW *win,
int line);
```

```
bool is_wintouched(WINDOW *win);

int touchline(WINDOW *win,
int start,
```

```
int count);

int touchwin(WINDOW *win);

int untouchwin(WINDOW *win);

int wtouchln(WINDOW *win,
int y,
int n,
int changed);
```

Description

The **touchline** subroutine touches the specified window (that is, marks it as having changed more recently than the last refresh operation). The **touchline** subroutine only touches count lines, beginning with line start.

The **untouchwin** subroutine marks all lines in the window as unchanged since the last refresh operation.

Calling the **wtouchln** subroutine, if changed is 1, touches n lines in the specified window, starting at line y. If changed is 0, **wtouchln** marks such lines as unchanged since the last refresh operation.

The **is_wintouchwin** subroutine determines whether the specified window is touched. The **is_linetouched** subroutine determines whether line line of the specified window is touched.

Parameters

Item	Description
line	
start	
count	
changed	
y	
n	
*win	

Return Values

The **is_linetouched** and **is_wintouched** subroutines return TRUE if any of the specified lines, or the specified window, respectively, has been touched since the last refresh operation. Otherwise, they return FALSE.

Upon successful completion, the other subroutines return OK. Otherwise, they return ERR. Exceptions to this are noted in the preceding subroutine.

Examples

For the **touchline** subroutine:

To set 10 lines for refresh starting from line 5 of the user-defined window `my_window`, use:

```
WINDOW *my_window;
touchline(my_window, 5, 10);
wrefresh(my_window);
```

This forces **curses** to disregard any optimization information it may have for lines 0-4 in `my_window`. **curses** assumes all characters in lines 0-4 have changed.

For the **touchwin** subroutine:

To refresh a user-defined parent window, `parent_window`, that has been edited through its subwindows, use:

```
WINDOW *parent_window;
touchwin(parent_window);

wrefresh(parent_window);
```

This forces **curses** to disregard any optimization information it may have for `my_window`. **curses** assumes all lines and columns have changed for `my_window`.

isalpha_l, isupper_l, islower_l, isdigit_l, isxdigit_l, isalnum_l, isspace_l, ispunct_l, isprint_l, isgraph_l, iscntrl_l, or isascii_ l Subroutines

Purpose

Classifies characters in the specified locale.

Library

Standard Character Library (**libc.a**)

Syntax

```
#include <ctype.h>
```

```
int isalpha_l (Character, locale);
int Character;
locale_t locale;
int isupper_l (Character, locale);
int Character;
locale_t locale;
int islower_l (Character, locale);
int Character;
locale_t locale;
int isdigit_l (Character, locale);
int Character;
locale_t locale;
int isxdigit_l (Character, locale);
int Character;
locale_t locale;
int isalnum_l (Character, locale);
int Character;
locale_t locale;
int isspace_l (Character, locale);
int Character;
locale_t locale;
int ispunct_l (Character, locale);
int Character;
locale_t locale;
int isprint_l (Character, locale);
int Character;
locale_t locale;
int isgraph_l (Character, locale);
int Character;
locale_t locale;
```

```
int iscntrl_l (Character, locale);
int Character;
locale_t locale;
```

Description

These routines are the same as the **isalpha**, **isupper**, **islower**, **isdigit**, **isxdigit**, **isalnum**, **isspace**, **ispunct**, **isprint**, **isgraph**, and **iscntrl** subroutines, except that they test the character C in the locale that is represented by locale instead of the current locale.

Return Codes

Refer to the **isupper** subroutine.

isblank, or isblank_l Subroutines

Purpose

Tests for a blank character.

Syntax

```
#include <ctype.h>

int isblank (c)
int c;
```

```
int isblank_l (c, Locale)
int c;
locale_t Locale;
```

Description

The **isblank** and **isblank_l** subroutines test whether the *c* parameter is a character of class **blank** in the program's current locale or in the locale represented by *Locale*.

The *c* parameter is a type **int**, the value of which the application shall ensure is a character representable as an **unsigned char** or equal to the value of the macro EOF. If the parameter has any other value, the behavior is undefined.

Parameters

Item	Description
c	Specifies the character to be tested.
Locale	Specifies the locale, in which the character is tested.

Return Values

The **isblank** and **isblank_l** subroutines return nonzero if *c* is a <blank>; otherwise, it returns 0.

isendwin Subroutine

Purpose

Determines whether the **endwin** subroutine was called without any subsequent refresh calls.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
isendwin()
```

Description

The **isendwin** subroutine determines whether the **endwin** subroutine was called without any subsequent refresh calls. If the **endwin** was called without any subsequent calls to the **wrefresh** or **doupdate** subroutines, the **isendwin** subroutine returns TRUE.

Return Values

Item	Description
TRUE	Indicates the **endwin** subroutine was called without any subsequent calls to the **wrefresh** or **doupdate** subroutines.
FALSE	Indicates subsequest calls to the refresh subroutines.

isfinite Macro

Purpose

Tests for finite value.

Syntax

```
#include <math.h>

int isfinite (x)
real-floating x;
```

Description

The **isfinite** macro determines whether its argument has a finite value (zero, subnormal, or normal, and not infinite or NaN). An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

Parameters

Item	Description
x	Specifies the value to be tested.

Return Values

The **isfinite** macro returns a nonzero value if its argument has a finite value.

isgreater Macro

Purpose

Tests if *x* is greater than *y*.

Syntax

```
#include <math.h>

int isgreater (x, y)
real-floating x;
real-floating y;
```

Description

The **isgreater** macro determines whether its first argument is greater than its second argument. The value of **isgreater**(*x, y*) is equal to (*x*) > (*y*); however, unlike (*x*) > (*y*), **isgreater**(*x, y*) does not raise the invalid floating-point exception when *x* and *y* are unordered.

Parameters

Item	Description
x	Specifies the first value to be compared.
y	Specifies the first value to be compared.

Return Values

Upon successful completion, the **isgreater** macro returns the value of (*x*) > (*y*).

If *x* or *y* is NaN, 0 is returned.

isgreaterequal Subroutine

Purpose

Tests if *x* is greater than or equal to *y*.

Syntax

```
#include <math.h>

int isgreaterequal (x, y)
real-floating x;
real-floating y;
```

Description

The **isgreaterequal** macro determines whether its first argument is greater than or equal to its second argument. The value of **isgreaterequal** (*x, y*) is equal to (*x*) >= (*y*); however, unlike (*x*) >= (*y*), **isgreaterequal** (*x, y*) does not raise the invalid floating-point exception when *x* and *y* are unordered.

Parameters

Item	Description
x	Specifies the first value to be compared.

Item	Description
y	Specifies the second value to be compared.

Return Values

Upon successful completion, the **isgreaterequal** macro returns the value of (*x*) >= (*y*).

If *x* or *y* is NaN, 0 is returned.

isinf Subroutine

Purpose

Tests for infinity.

Syntax

```
#include <math.h>

int isinf (x)
real-floating x;
```

Description

The **isinf** macro determines whether its argument value is an infinity (positive or negative). An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

Parameters

Item	Description
x	Specifies the value to be checked.

Return Values

The **isinf** macro returns a nonzero value if its argument has an infinite value.

isless Macro

Purpose

Tests if *x* is less than *y*.

Syntax

```
#include <math.h>
int isless (x, y)
real-floating x;
real-floating y;
```

Description

The **isless** macro determines whether its first argument is less than its second argument. The value of **isless**(*x, y*) is equal to (*x*) < (*y*); however, unlike (*x*) < (*y*), **isless**(*x, y*) does not raise the invalid floating-point exception when *x* and *y* are unordered.

Parameters

Item	Description
x	Specifies the first value to be compared.
y	Specifies the second value to be compared.

Return Values

Upon successful completion, the **isless** macro returns the value of (*x*) < (*y*).

If *x* or *y* is NaN, 0 is returned.

islessequal Macro

Purpose

Tests if *x* is less than or equal to *y*.

Syntax

```
#include <math.h>

int islessequal (x, y)
real-floating x;
real-floating y;
```

Description

The **islessequal** macro determines whether its first argument is less than or equal to its second argument. The value of **islessequal**(*x, y*) is equal to (*x*) <= (*y*); however, unlike (*x*) <= (*y*), **islessequal**(*x, y*) does not raise the invalid floating-point exception when *x* and *y* are unordered.

Parameters

Item	Description
x	Specifies the first value to be compared.
y	Specifies the second value to be compared.

Return Values

Upon successful completion, the **islessequal** macro returns the value of (*x*) <= (*y*).

If *x* or *y* is NaN, 0 is returned.

islessgreater Macro

Purpose

Tests if *x* is less than or greater than *y*.

Syntax

```
#include <math.h>

int islessgreater (x, y)
```

```
real-floating x;
real-floating y;
```

Description

The **islessgreater** macro determines whether its first argument is less than or greater than its second argument. The **islessgreater**(*x, y*) macro is similar to (*x*) < (*y*) || (*x*) > (*y*); however, **islessgreater**(*x, y*) does not raise the invalid floating-point exception when *x* and *y* are unordered (nor does it evaluate *x* and *y* twice).

Parameters

Item	Description
x	Specifies the first value to be compared.
y	Specifies the second value to be compared.

Return Values

Upon successful completion, the **islessgreater** macro returns the value of (*x*) < (*y*) || (*x*) > (*y*).

If *x* or *y* is NaN, 0 is returned.

isnormal Macro

Purpose

Tests for a normal value.

Syntax

```
#include <math.h>

int isnormal (x)
real-floating x;
```

Description

The **isnormal** macro determines whether its argument value is normal (neither zero, subnormal, infinite, nor NaN) or not. An argument represented in a format wider than its semantic type is converted to its semantic type. Determination is based on the type of the argument.

Parameters

Item	Description
x	Specifies the value to be tested.

Return Values

The **isnormal** macro returns a nonzero value if its argument has a normal value.

isunordered Macro

Purpose

Tests if arguments are unordered.

Syntax

```
#include <math.h>
int isunordered (x, y)
real-floating x;
real-floating y;
```

Description

The **isunordered** macro determines whether its arguments are unordered.

Parameters

Item	Description
x	Specifies the first value in the order.
y	Specifies the second value in the order.

Return Values

Upon successful completion, the **isunordered** macro returns 1 if its arguments are unordered, and 0 otherwise.

If *x* or *y* is NaN, 0 is returned.

iswalnum, iswalpha, iswcntrl, iswdigit, iswgraph, iswlower, iswprint, iswpunct, iswspace, iswupper, or iswxdigit Subroutine

Purpose

Tests a wide character for membership in a specific character class.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
int iswalnum (WC)
wint_t WC;
```

```
int iswalpha (WC)
wint_t WC;
```

```
int iswcntrl (WC)
wint_t WC;
```

```
int iswdigit (WC)
wint_t WC;
```

```
int iswgraph (WC)
wint_t WC;
```

```
int iswlower (WC)
wint_t WC;
```

```
int iswprint (WC)
wint_t WC;
```

```
int iswpunct (WC)
wint_t WC;
```

```
int iswspace (WC)
wint_t WC;
```

```
int iswupper (WC)
wint_t WC;
```

```
int iswxdigit (WC)
wint_t WC;
```

Description

The **isw** subroutines check the character class status of the wide character code specified by the *WC* parameter. Each subroutine tests to see if a wide character is part of a different character class. If the wide character is part of the character class, the **isw** subroutine returns true; otherwise, it returns false.

Each subroutine is named by adding the **isw** prefix to the name of the character class that the subroutine tests. For example, the **iswalpha** subroutine tests whether the wide character specified by the *WC* parameter is an alphabetic character. The character classes are defined as follows:

Item	Description
alnum	Alphanumeric character.
alpha	Alphabetic character.
cntrl	Control character. No characters in the **alpha** or **print** classes are included.
digit	Numeric digit character.
graph	Graphic character for printing, not including the space character or **cntrl** characters. Includes all characters in the **digit** and **punct** classes.
lower	Lowercase character. No characters in **cntrl**, **digit**, **punct**, or **space** are included.
print	Print character. All characters in the **graph** class are included, but no characters in **cntrl** are included.
punct	Punctuation character. No characters in the **alpha**, **digit**, or **cntrl** classes, or the space character are included.
space	Space characters.
upper	Uppercase character.
xdigit	Hexadecimal character.

Parameters

Item	Description
WC	Specifies a wide character for testing.

Return Values

If the wide character tested is part of the particular character class, the **isw** subroutine returns a nonzero value; otherwise it returns a value of 0.

iswalnum_l, iswalpha_l, iswcntrl_l, iswdigit_l, iswgraph_l, iswlower_l, iswprint_l, iswpunct_l, iswspace_l, iswupper_l, or iswxdigit_l Subroutines

Purpose

Tests a wide character for membership in a specific character class.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
int iswalnum_l (WC, locale)
wint_t WC;
locale_t locale;
int iswalpha_l (WC, locale)
wint_t WC;
locale_t locale;
int iswcntrl_l (WC, locale)
wint_t WC;
locale_t locale;
int iswdigit_l (WC, locale)
wint_t WC;
locale_t locale;
int iswgraph_l (WC, locale)
wint_t WC;
locale_t locale;
int iswlower_l (WC, locale)
wint_t WC;
locale_t locale;
int iswprint_l (WC, locale)
wint_t WC;
locale_t locale;
int iswpunct_l (WC, locale)
wint_t WC;
locale_t locale;
int iswspace_l (WC, locale)
wint_t WC;
locale_t locale;
int iswupper_l (WC, locale)
wint_t WC;
locale_t locale;
int iswxdigit_l (WC, locale)
wint_t WC;
locale_t locale;
```

Description

These routines are the same as the **iswalnum**, **iswalpha**, **iswcntrl** , **isdigit**, **iswgraph**, **iswlower**, **iswprint**, **iswpunct**, **iswspace**, **iswupper**, and **iswxdigit** subroutines, except that they test the character WC in the locale that is represented by locale instead of the current locale.

Return Codes

Refer to the **iswupper** subroutine.

iswblank, or iswblank_l Subroutines

Purpose

Tests for a blank wide-character code.

Syntax

```
#include <wctype.h>

int iswblank (wc)
wint_t wc;
```

```
int iswblank_l(wc, Locale)
wint_t wc;
locale_t Locale;
```

Description

The **iswblank** and **iswblank_l** subroutines test whether the *wc* parameter is a wide-character code representing a character of class **blank** in the program's current locale or in the locale represented by *Locale*.

The *wc* parameter is a **wint_t**, the value of which the application ensures is a wide-character code corresponding to a valid character in the current locale, or equal to the value of the macro **WEOF**. If the parameter has any other value, the behavior is undefined.

Parameters

Item	Description
wc	Specifies the value to be tested.
Locale	Specifies the locale, in which the character is tested.

Return Values

The **iswblank** and **iswblank_l** subroutines return a nonzero value if the *wc* parameter is a blank wide-character code; otherwise, it returns a 0.

iswctype, iswctype_l or is_wctype Subroutine

Purpose

Determines properties of a wide character.

Library

Standard C Library (**libc. a**)

Syntax

```
#include  <wchar.h>
```

```
int iswctype  ( WC,  Property)
wint_t WC;
wctype_t Property;
```

```
int iswctype_l  ( WC, Property, Locale)
wint_t WC;
wctype_t Property;
locale_t Locale;
```

```
int is_wctype ( WC,  Property)
wint_t WC;
wctype_t Property;
```

Description

The **iswctype**, and **iswctype_l** subroutines test the wide character specified by the *WC* parameter to determine if it has the property specified by the *Property* parameter. The **iswctype**, and **iswctype_l** subroutines are defined for the wide-character null value and for values in the character range of the current code set, defined in the current locale or in the locale represented by *Locale*. The **is_wctype** subroutine is identical to the **iswctype** subroutine.

The **iswctype** subroutine adheres to X/Open Portability Guide Issue 5.

Parameters

Item	Description
WC	Specifies the wide character to be tested.
Property	Specifies the property for which to test.
Locale	Specifies the locale, in which the character is tested.

Return Values

If the *WC* parameter has the property specified by the *Property* parameter, the **iswctype**, and **iswctype_l** subroutines return a nonzero value. If the value specified by the *WC* parameter does not have the property specified by the *Property* parameter, the **iswctype**, and **iswctype_l** subroutines return a value of zero. If the value specified by the *WC* parameter is not in the subroutine's domain, the result is undefined. If the value specified by the *Property* parameter is not valid (that is, not obtained by a call to the **wctype** subroutine, or the *Property* parameter has been invalidated by a subsequent call to the **setlocale** subroutine that has affected the **LC_CTYPE** category), the result is undefined.

j

The following Base Operating System (BOS) runtime services begin with the letter *j*.

jcode Subroutines

Purpose

Perform string conversion on 8-bit processing codes.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <jcode.h>
```

```
char *jistosj(String1, String2)
char *String1, *String2;
```

```
char *jistouj(String1, String2)
char *String1, *String2;
```

```
char *sjtojis(String1, String2)
char *String1, *String2;
```

```
char *sjtouj(String1, String2)
char *String1, *String2;
```

```
char *ujtojis(String1, String2)
char *String1, *String2;
```

```
char *ujtosj(String1, String2)
char *String1, *String2;
```

```
char *cjistosj(String1, String2)
char *String1, *String2;
```

```
char *cjistouj(String1, String2)
char *String1, *String2;
```

```
char *csjtojis(String1, String2)
char *String1, *String2;
```

```
char *csjtouj(String1, String2)
char *String1, *String2;
```

```
char *cujtojis(String1, String2)
char *String1, *String2;
```

```
char *cujtosj(String1, String2)
char *String1, *String2;
```

Description

The **jistosj**, **jistouj**, **sjtojis**, **sjtouj**, **ujtojis**, and **ujtosj** subroutines perform string conversion on 8-bit processing codes. The *String2* parameter is converted and the converted string is stored in the *String1* parameter. The overflow of the *String1* parameter is not checked. Also, the *String2* parameter must be a valid string. Code validation is not permitted.

The **jistosj** subroutine converts JIS to SJIS. The **jistouj** subroutine converts JIS to UJIS. The **sjtojis** subroutine converts SJIS to JIS. The **sjtouj** subroutine converts SJIS to UJIS. The **ujtojis** subroutine converts UJIS to JIS. The **ujtosj** subroutine converts UJIS to SJIS.

The **cjistosj**, **cjistouj**, **csjtojis**, **csjtouj**, **cujtojis**, and **cujtosj** macros perform code conversion on 8-bit processing JIS Kanji characters. A character is removed from the *String2* parameter, and its code is converted and stored in the *String1* parameter. The *String1* parameter is returned. The validity of the *String2* parameter is not checked.

The **cjistosj** macro converts from JIS to SJIS. The **cjistouj** macro converts from JIS to UJIS. The **csjtojis** macro converts from SJIS to JIS. The **csjtouj** macro converts from SJIS to UJIS. The **cujtojis** macro converts from UJIS to JIS. The **cujtosj** macro converts from UJIS to SJIS.

Parameters

Item	Description
String1	Stores converted string or code.
String2	Stores string or code to be converted.

Japanese conv Subroutines

Purpose

Translates predefined Japanese character classes.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ctype.h>
int atojis (Character)
int Character;
```

```
int jistoa (Character)
int Character;
```

```
int _atojis (Character)
int Character;
```

```
int _jistoa (Character)
int Character;
```

```
int tojupper (Character)
int Character;
```

```
int tojlower (Character)
int Character;
```

```
int _tojupper (Character)
int Character;
```

```
int _tojlower (Character)
int Character;
```

```
int toujis (Character)
int Character;
```

```
int kutentojis (Character)
int Character;
```

```
int tojhira (Character)
int Character;
```

```
int tojkata (Character)
int Character;
```

Description

When running the operating system with Japanese Language Support on your system, the legal value of the *Character* parameter is in the range from 0 to **NLCOLMAX**.

The **jistoa** subroutine converts an SJIS ASCII equivalent to the corresponding ASCII equivalent. The **atojis** subroutine converts an ASCII character to the corresponding SJIS equivalent. Other values are returned unchanged.

The **_jistoa** and **_atojis** routines are macros that function like the **jistoa** and **atojis** subroutines, but are faster and have no error checking function.

The **tojlower** subroutine converts a SJIS uppercase letter to the corresponding SJIS lowercase letter. The **tojupper** subroutine converts an SJIS lowercase letter to the corresponding SJIS uppercase letter. All other values are returned unchanged.

The **_tojlower** and **_tojupper** routines are macros that function like the **tojlower** and **tojupper** subroutines, but are faster and have no error-checking function.

The **toujis** subroutine sets all parameter bits that are not 16-bit SJIS code to 0.

The **kutentojis** subroutine converts a kuten code to the corresponding SJIS code. The **kutentojis** routine returns 0 if the given kuten code is invalid.

The **tojhira** subroutine converts an SJIS katakana character to its SJIS hiragana equivalent. Any value that is not an SJIS katakana character is returned unchanged.

The **tojkata** subroutine converts an SJIS hiragana character to its SJIS katakana equivalent. Any value that is not an SJIS hiragana character is returned unchanged.

The **_tojhira** and **_tojkata** subroutines attempt the same conversions without checking for valid input.

For all functions except the **toujis** subroutine, the out-of-range parameter values are returned without conversion.

Parameters

Item	Description
Character	Character to be converted.
Pointer	Pointer to the escape sequence.
CharacterPointer	Pointer to a single **NLchar** data type.

Japanese ctype Subroutines

Purpose

Classify characters.

Library

Standard Character Library (**libc.a**)

Syntax

```
#include <ctype.h>
```

```
int isjalpha ( Character)
int Character;
```

```
int isjupper (Character)
int Character;
```

```
int isjlower (Character)
int Character;
```

```
int isjlbytekana (Character)
int Character;
```

```
int isjdigit (Character)
int Character;
```

```
int isjxdigit (Character)
int Character;
```

```
int isjalnum (Character)
int Character;
```

```
int isjspace (Character)
int Character;
```

```
int isjpunct (Character)
int Character;
```

```
int isjparen (Character)
int Character;
```

```
int isparent (Character)
intCharacter;
```

```
int isjprint (Character)
int Character;
```

```
int isjgraph (Character)
int Character;
```

```
int isjis (Character)
int Character;
```

```
int isjhira (wc)
wchar_t wc;
```

```
int isjkanji (wc)
wchar_wc;
```

```
int isjkata (wc)
wchar_t wc;
```

Description

The **Japanese ctype** subroutines classify character-coded integer values specified in a table. Each of these subroutines returns a nonzero value for True and 0 for False.

Parameters

Item	Description
Character	Character to be tested.

Return Values

The **isjprint** and **isjgraph** subroutines return a 0 value for user-defined characters.

k

The following Base Operating System (BOS) runtime services begin with the letter *k*.

keyname, key_name Subroutine

Purpose

Gets the name of keys.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

char *keyname(int c);

char *key_name(wchar_t c);
```

Description

The **keyname** and **key_name** subroutines generate a character string whose value describes the key *c*. The *c* argument of **keyname** can be an 8-bit character or a key code. The *c* argument of **key_name** must be a wide character.

The string has a format according to the first applicable row in the following table:

Item	Description
Input	Format of Returned String
Visible character	The same character
Control character	^X
Meta-character (**keyname** only)	M-X
Key value defined in <curses.h> (**keyname** only)	KEY_name
None of the above	UNKNOWN KEY

The meta-character notation shown above is used only, if meta-characters are enabled.

Parameter

c

Return Values

Upon successful completion, the **keyname** subroutine returns a pointer to a string as described above, Otherwise, it returns a null pointer.

Examples

```
int key;
char *name;
```

```
keypad(stdscr, TRUE);
addstr("Hit a key");
key=getch();
name=keyname(key);
```

Note: If the Page Up key is pressed, keyname will return **KEY_PPAGE**.

keypad Subroutine

Purpose

Enables or disables abbreviation of function keys.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

int keypad(WINDOW **win*,
bool *bf*);

Description

The **keypad** subroutine controls keypad translation. If *bf* is TRUE, keypad translation is turned on. If *bf* is FALSE, keypad translation is turned off. The initial state is FALSE.

This subroutine affects the behavior of any function that provides keyboard input.

If the terminal in use requires a command to enable it to transmit distinctive codes when a function key is pressed, then after keypad translation is first enabled, the implemenation transmits this command to the terminal before an affected input function tries to read any characters from that terminal.

Parameters

Item	Description
bf	
**win*	Specifies the window in which to enable or disable the keypad.

Return Values

Upon successful completion, the **keypad** subroutine returns OK**.** Otherwise, it returns ERR.

Examples

To turn on the keypad in the user-defined window `my_window`, use:

```
WINDOW *my_window;
keypad(my_window, TRUE);
```

killchar or killwchar Subroutine

Purpose

Terminal environment query functions.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
char killchar(void);
```

```
int killwchar(wchar_t *ch);
```

Description

The **killchar** subroutine returns the current line.

The **killchar** subroutine stores the current line kill character in the object pointed to by *ch*. If no line kill character has been defined, the subroutine will fail and the object pointed to by *ch* will not be changed.

Parameters

**ch*

Return Values

The **killchar** subroutine returns the line kill character. The return value is unspecified when this character is a multi-byte character.

Upon successful completion, the **killchar** subroutine returns OK. Otherwise, it returns ERR.

kget_proc_info Kernel Service

Purpose

Allows a kernel extension to get information about a process or process group.

Syntax

```
#include <procinfo.h>
```

kerrno_t kget_proc_info (*cmd*,*id*,*data*,*size***)**
int *cmd***;**
pid_t *id***;**
void * *data***;**
size_t * *size***;**

Parameters

Item	Description
cmd	Command indicating data to be returned.
id	Process group ID (PID) for which the information is retrieved.
data	Data region that contains the data returned
size	Size of the data region

Description

The **kget_proc_info** kernel service retrieves information about a process or process group for a kernel extension. The following **cmd** values are supported, with the specified parameters and return codes:

Parameter	Return Codes
VALIDATE_PID	This command determines if a PID or process group ID is valid. The *data* and *size* parameters are unused. This command will return **0** if the PID is valid, and ESRCH_INVALID_PID if it is not valid.
GET_PROCENTRY64	This command fills in a **procentry64** structure for the given PID. The data should point to a **struct procentry64** and size should be the size of a **struct procentry64**. This command will return 0 on success, EINVAL_NULL_SIZE for a NULL size parameter, EINVAL_NULL_DATA for a NULL data parameter, ESRCH_INVALID_PID if the PID is invalid, ERANGE_INSUFFICIENT_SIZE if size is insufficient to contain the **struct procentry64**, and EPERM_INSUFFICIENT_PRIVS if the current process is not allowed to obtain information about the target process.
GET_PGRP and **GET_PGRP_BY_MEMBER**	These commands fill in an array of PIDs in a process group. The process group is specified either by a process group PID (GET_PGRP) or the PID of a member of the process group (GET_PGRP_BY_MEMBER). If the *data* parameter is NULL, this will update the target *size* parameter with the size needed to hold all the PIDs. On successful return, the *data* parameter is filled with an array of PIDs and the *size* parameter is filled in with the actual size used. A value of 0 is returned for success. This command will return EINVAL_NULL_SIZE for a NULL *size* parameter, ESRCH_INVALID_PID if the PID is invalid, and ERANGE_INSUFFICIENT_SIZE if a *data* parameter is specified and *size* is insufficient to contain the array of PIDs. If the size is insufficient, the *size* parameter is updated with the correct needed size. **Note:** While the data returned is consistent during the call, on return, the process or process group may change. Specifically, the size needed to hold the array of PIDs may be insufficient on a successive call.

Execution Environment

kget_proc_info must be called from the process environment only.

Return Values

Upon successful completion, 0 is returned. If the call is unsuccessful, an error number is returned as detailed in the corresponding command. Additionally, EINVAL_INVALID_COMMAND is returned for an invalid command.

kill or killpg Subroutine

Purpose

Sends a signal to a process or to a group of processes.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <signal.h>
```

```
int kill(
 Process,
 Signal)
pid_t Process;
int Signal;
```

```
killpg(
 ProcessGroup, Signal)
int ProcessGroup, Signal;
```

Description

The **kill** subroutine sends the signal specified by the *Signal* parameter to the process or group of processes specified by the *Process* parameter.

To send a signal to another process, either the real or the effective user ID of the sending process must match the real or effective user ID of the receiving process, and the calling process must have root user authority.

The processes that have the process IDs of 0 and 1 are special processes and are sometimes referred to here as *proc0* and *proc1*, respectively.

Processes can send signals to themselves.

Note: Sending a signal does not imply that the operation is successful. All signal operations must pass the access checks prescribed by each enforced access control policy on the system.

The following interface is provided for BSD Compatibility:

```
killpg(ProcessGroup, Signal)
int ProcessGroup; Signal;
```

This interface is equivalent to:

```
if (ProcessGroup < 0)
{
  errno = ESRCH;
  return (-1);
}
return (kill(-ProcessGroup, Signal));
```

Parameters

Item	Description
Process	Specifies the ID of a process or group of processes. If the *Process* parameter is greater than 0, the signal specified by the *Signal* parameter is sent to the process identified by the *Process* parameter. If the *Process* parameter is 0, the signal specified by the *Signal* parameter is sent to all processes, excluding *proc0* and *proc1*, whose process group ID matches the process group ID of the sender. If the value of the *Process* parameter is a negative value other than -1 and if the calling process passes the access checks for the process to be signaled, the signal specified by the *Signal* parameter is sent to all the processes, excluding *proc0* and *proc1*. If the user ID of the calling process has root user authority, all processes, excluding *proc0* and *proc1*, are signaled. If the value of the *Process* parameter is a negative value other than -1, the signal specified by the *Signal* parameter is sent to all processes having a process group ID equal to the absolute value of the *Process* parameter. If the value of the *Process* parameter is -1, the signal specified by the *Signal* parameter is sent to all processes which the process has permission to send that signal.
Signal	Specifies the signal. If the Signal parameter is a null value, error checking is performed but no signal is sent. This parameter is used to check the validity of the *Process* parameter.
ProcessGroup	Specifies the process group.

Return Values

Upon successful completion, the **kill** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **kill** subroutine is unsuccessful and no signal is sent if one or more of the following are true:

Item	Description
EINVAL	The *Signal* parameter is not a valid signal number.
EINVAL	The *Signal* parameter specifies the **SIGKILL**, **SIGSTOP**, **SIGTSTP**, or **SIGCONT** signal, and the *Process* parameter is 1 (*proc1*).
ESRCH	No process can be found corresponding to that specified by the *Process* parameter.
EPERM	The real or effective user ID does not match the real or effective user ID of the receiving process, or else the calling process does not have root user authority.

kleenup Subroutine

Purpose

Cleans up the run-time environment of a process.

Library

Syntax

```
int kleenup( FileDescriptor,  SigIgn,  SigKeep)
int FileDescriptor;
int SigIgn[ ];
int SigKeep[ ];
```

Description

The **kleenup** subroutine cleans up the run-time environment for a trusted process by:

- Closing unnecessary file descriptors.
- Resetting the alarm time.
- Resetting signal handlers.
- Clearing the value of the **real directory read** flag described in the **ulimit** subroutine.
- Resetting the **ulimit** value, if it is less than a reasonable value (8192).

Parameters

Item	Description
FileDescriptor	Specifies a file descriptor. The **kleenup** subroutine closes all file descriptors greater than or equal to the *FileDescriptor* parameter.
SigIgn	Points to a list of signal numbers. If these are nonnull values, this list is terminated by 0s. Any signals specified by the *SigIgn* parameter are set to **SIG_IGN**. The handling of all signals not specified by either this list or the *SigKeep* list are set to **SIG_DFL**. Some signals cannot be reset and are left unchanged.
SigKeep	Points to a list of signal numbers. If these are nonnull values, this list is terminated by 0s. The handling of any signals specified by the *SigKeep* parameter is left unchanged. The handling of all signals not specified by either this list or the *SigIgn* list are set to **SIG_DFL**. Some signals cannot be reset and are left unchanged.

Return Values

The **kleenup** subroutine is always successful and returns a value of 0. Errors in closing files are not reported. It is not an error to attempt to modify a signal that the process is not allowed to handle.

knlist Subroutine

Purpose

Translates names to addresses in the running system.

Syntax

```
#include <nlist.h>
```

```
int knlist( NList,  NumberOfElements,  Size)
struct nlist *NList;
int NumberOfElements;
int Size;
```

Description

The **knlist** subroutine allows a program to look up the addresses of symbols exported by the kernel and kernel extensions.

The n_name field in the **nlist** structure specifies the name of a symbol for which the address is requested. If the symbol is found, its address is saved in the **n_value** field, and the remaining fields are not modified. If the symbol is not found, all fields, other than **n_name**, are set to 0.

In a 32-bit program, the **n_value** field is a 32-bit field, which is too small for some kernel addresses. To allow the addresses of all specified symbols to be obtained, 32-bit programs must use the **nlist64** structure, which contains a 64-bit **n_value** field. For example, if NList64 is the address of an array of **nlist64** structures, the **knlist** subroutine can be called as shown in the following example:

```
rc = knlist((struct nlist *)Nlist64,
                  NumberOfElements,
                  sizeof(structure nlist64));
```

The **nlist** and **nlist64** structures include the following fields:

Item	Description
`char *n_name`	Specifies the name of the symbol for which the address is to be retrieved.
`long n_value`	The address of the symbol, filled in by the **knlist** subroutine. This field is included in the **nlist** structure.
`long long n_value`	The address of the symbol, filled in by the **knlist** subroutine. This field is included in the **nlist64** structure.

The **nlist.h** file is automatically included by the **a.out.h** file for compatibility. However, do not include the **a.out.h** file if you only need the information necessary to use the **knlist** subroutine. If you do include the **a.out.h** file, follow the **#include** statement with the following line:

```
#undef n_name
```

Note:

1. If both the **nlist.h** and **netdb.h** files are to be included, the **netdb.h** file should be included before the **nlist.h** file in order to avoid a conflict with the **n_name** structure member. Likewise, if both the **a.out.h** and **netdb.h** files are to be included, the **netdb.h** file should be included before the **a.out.h** file to avoid a conflict with the **n_name** structure.
2. If the **netdb.h** file and either the **nlist.h** or **syms.h** file are included, the n_name field will be defined as _n._n_name. This definition allows you to access the n_name field in the **nlist** or **syment** structure. If you need to access the n_name field in the **netent** structure, undefine the n_name field by entering:

   ```
   #undef n_name
   ```

   before accessing the n_name field in the **netent** structure. If you need to access the n_name field in a **syment** or **nlist** structure after undefining it, redefine the n_name field with:

   ```
   #define n_name _n._n_name
   ```

Parameters

Item	Description
NList	Points to an array of **nlist** or **nlist64** structures.
NumberOfElements	Specifies the number of structures in the array of **nlist** or **nlist64** structures.
Size	Specifies the size of each structure. The only allowed values are `sizeof(struct nlist)` or `sizeof(struct nlist64)`.

Return Values

Upon successful completion, the **knlist** subroutine returns a value of 0. Otherwise, a value of -1 is returned, and the **errno** variable is set to indicate the error.

Error Codes

The **knlist** subroutine fails when one of the following is true:

Item	Description
EINVAL	The *NumberOfElements* parameters is less than 1 or the *Size* parameter is neither `sizeof(struct nlist)` nor `sizeof(struct nlist64)`.
EFAULT	The *NList* parameter is not a valid address. One or more symbols in the array specified by the *Nlist* parameter were not found. The address of one of the symbols does not fit in the **n_value** field. This is only possible if the caller is a 32-bit program and the *Size* parameter is `sizeof(struct nlist))`.

kpidstate Subroutine

Purpose

Returns the status of a process.

Syntax

```
kpidstate (pid)
pid_t pid;
```

Description

The **kpidstate** subroutine returns the state of a process specified by the *pid* parameter. The **kpidstate** subroutine can only be called by a process.

Parameters

Item	Description
pid	Specifies the product ID.

Return Values

If the *pid* parameter is not valid, KP_NOTFOUND is returned. If the pid parameter is valid, the following settings in the process state determine what is returned:

Item	Description
SNONE	Return KP_NOTFOUND.
SIDL	Return KP_INITING.
SZOMB	Return KP_EXITING, also if SEXIT in pv_flag.
SSTOP	Return KP_STOPPED.

Otherwise the pid is alive and KP_ALIVE is returned.

Error Codes

l

The following Base Operating System (BOS) runtime services begin with the letter *l*.

_lazySetErrorHandler Subroutine

Purpose

Installs an error handler into the lazy loading runtime system for the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/ldr.h>
#include <sys/errno.h>
```

```
typedef void (*_handler_t(
char *_module,
char *_symbol,
unsigned int _errVal ))();
```

```
handler_t *_lazySetErrorHandler(err_handler)
handler_t *err_handler;
```

Description

This function allows a process to install a custom error handler to be called when a lazy loading reference fails to find the required module or function. This function should only be used when the main program or one of its dependent modules was linked with the **-blazy** option. To call **_lazySetErrorHandler** from a module that is not linked with the **-blazy** option, you must use the **-lrtl** option. If you use **-blazy**, you do not need to specify **-lrtl**.

This function is not thread safe. The calling program should ensure that **_lazySetErrorHandler** is not called by multiple threads at the same time.

The user-supplied error handler may print its own error message, provide a substitute function to be used in place of the called function, or call the **longjmp** subroutine. To provide a substitute function that will be called instead of the originally referenced function, the error handler should return a pointer to the substitute function. This substitute function will be called by all subsequent calls to the intended function from the same module. If the value returned by the error handler appears to be invalid (for example, a NULL pointer), the default error handler will be used.

Each calling module resolves its lazy references independent of other modules. That is, if module A and B both call **foo** subroutine in module C, but module C does not export **foo** subroutine, the error handler will be called once when **foo** subroutine is called for the first time from A, and once when **foo** subroutine is called for the first time from B.

The default lazy loading error handler will print a message containing: the name of module that the program required; the name of the symbol being accessed; and the error value generated by the failure. Since the default handler considers a lazy load error to be fatal, the process will exit with a status of 1.

During execution of a program that utilizes lazy loading, there are a few conditions that may cause an error to occur. In all cases the current error handler will be called.

1. The referenced module (which is to be loaded upon function invocation) is unavailable or cannot be loaded. The *errVal* parameter will probably indicate the reason for the error if a system call failed.
2. A function is referenced, but the loaded module does not contain a definition for the function. In this case, *errVal* parameter will be **EINVAL**.

Some possibilities as to why either of these errors might occur:

1. The **LIBPATH** environment variable may contain a set of search paths that cause the application to load the wrong version of a module.
2. A module has been changed and no longer provides the same set of symbols that it did when the application was built.
3. The **load** subroutine fails due to a lack of resources available to the process.

Parameters

Item	Description
err_handler	A pointer to the new error handler function. The new function should accept 3 arguments: ***module*** The name of the referenced module. ***symbol*** The name of the function being called at the time the failure occurred. ***errVal*** The value of **errno** at the time the failure occurred, if a system call used to load the module fails. For other failures, errval may be **EINVAL** or **ENOMEM**.

Note that the value of module or symbol may be NULL if the calling module has somehow been corrupted.

If the *err_handler* parameter is NULL, the default error handler is restored.

Return Value

The function returns a pointer to the previous user-supplied error handler, or NULL if the default error handler was in effect.

l3tol or ltol3 Subroutine

Purpose

Converts between 3-byte integers and long integers.

Library

Standard C Library (**libc.a**)

Syntax

```
void l3tol ( LongPointer,  CharacterPointer,  Number)
long *LongPointer;
char *CharacterPointer;
int Number;
```

```
void ltol3 (CharacterPointer, LongPointer, Number)
char *CharacterPointer;
long *LongPointer;
int Number;
```

Description

The **l3tol** subroutine converts a list of the number of 3-byte integers specified by the *Number* parameter packed into a character string pointed to by the *CharacterPointer* parameter into a list of long integers pointed to by the *LongPointer* parameter.

The **ltol3** subroutine performs the reverse conversion, from long integers (the *LongPointer* parameter) to 3-byte integers (the *CharacterPointer* parameter).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

Parameters

Item	Description
LongPointer	Specifies the address of a list of long integers.
CharacterPointer	Specifies the address of a list of 3-byte integers.
Number	Specifies the number of list elements to convert.

l64a_r Subroutine

Purpose

Converts base-64 long integers to strings.

Library

Thread-Safe C Library (**libc_r.a**)

Syntax

```
#include <stdlib.h>

int l64a_r (Convert, Buffer, Length)
long  Convert;
char * Buffer;
int  Length;
```

Description

The **l64a_r** subroutine converts a given long integer into a base-64 string.

Programs using this subroutine must link to the **libpthreads.a** library.

For base-64 characters, the following ASCII characters are used:

Character	Description
.	Represents 0.
/	Represents 1.
0 -9	Represents the numbers 2-11.
A-Z	Represents the numbers 12-37.
a-z	Represents the numbers 38-63.

The **l64a_r** subroutine places the converted base-64 string in the buffer pointed to by the *Buffer* parameter.

Parameters

Item	Description
Convert	Specifies the long integer that is to be converted into a base-64 ASCII string.
Buffer	Specifies a working buffer to hold the converted long integer.
Length	Specifies the length of the *Buffer* parameter.

Return Values

Item	Description
0	Indicates that the subroutine was successful.
-1	Indicates that the subroutine was not successful. If the **l64a_r** subroutine is not successful, the **errno** global variable is set to indicate the error.

Error Codes

If the **l64a_r** subroutine is not successful, it returns the following error code:

Item	Description
EINVAL	The *Buffer* parameter value is invalid or too small to hold the resulting ASCII string.

labelsession Subroutine

Purpose

Determines user access to system by validating the user security labels against the system labels.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>

int labelsession (Name, Mode, TTY, EffSL, EffTL, Msg [, Flag])
char *Name;
intMode;
char *TTY;
char *EffSL;
char *EffTL;
char **Msg;
int Flag;
```

Description

The **labelsession** subroutine determines whether the user specified by the *Name* parameter is allowed to access the system based on the sensitivity and the integrity clearances of the user. The *Mode* parameter gives the mode of the account usage and the *TTY* parameter defines the terminal that is used for access. The *EffSL* and *EffTL* parameters specify the effective sensitivity label and the effective integrity label for the session respectively. The *Msg* parameter returns an information message that explains the reason that the subroutine fails.

The **labelsession** subroutine fails under the following circumstances:

- The *Mode* parameter is not S_SU and user ID of the user is less than 128. Any user with a user ID (uid) less than 128 is only allowed to login with the **su** command.
- Either the sensitivity labels or the integrity labels, or both labels are not properly dominated.
- The specified effective SL is not within the user's clearance range and the user does not have the **aix.mls.label.outsideaccred** authority.
- The effective SL of the user is not in the TTY's label range.
- The specified effective TL is not in the user's clearance range.
- If the TTY has a TL set, the specified effective TL is not equal to the TTY's TL.
- The *Flag* parameter is not specified for S_SU and the current user's label does not dominate those of the new users.

Restriction: This subroutine is applicable only on a Trusted AIX system.

Parameters

Item	Description
Name	Specifies the user login name.
Mode	Specifies the mode to use. The *Mode* parameter contains one of the following valid values that are defined in the **login.h** file: **S_LOGIN** Local login **S_RLOGIN** Remote login using the **rlogind** and **telnetd** commands **S_SU** Login in using the **su** command **S_FTP** FTP based login
TTY	Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no TTY checking is done.
EffSL	Specifies the effective SL that the session requires.
EffTL	Specifies the effective TL that the session requires.
Msg	Returns a message to the user interface that explains the reason why the subroutine fails. The returned value is either a pointer to a valid string within memory allocated storage or a null value.
Flag	When the *Flag* parameter is set to 1, the current user labels do not need to dominate those of the new user to allow access. This parameter is valid only for the S_SU mode. This parameter is ignored for all other session types.

Security

Access Control: The calling process must have access to the account information in the user database and the port information in the port database. The calling process must also have the privileges that are required by the subroutines that this subroutine invokes.

File Accessed

Mode	File
r	**/etc/security/enc/LabelEncodings**
r	**/etc/security/user**

Return Values

If the session labels are valid for the specified usage, the **labelsession** subroutine returns a value of zero. Otherwise, the subroutine returns a value of -1, sets the **errno** global value and the *Msg* parameter returns the error information.

Error Codes

If the subroutine fails, it returns one of the following error codes:

Item	Description
EINVAL	Error in label encodings file or error in the label dominance
EINVAL	The specified effective SL is not valid on the system
ENOATTR	The clearance attributes for the user do not exist
ENOMEM	Memory cannot be allocated to store the returned value
EPERM	No permission to complete the operation

LAPI_Addr_get Subroutine

Purpose

Retrieves a function address that was previously registered using LAPI_Addr_set.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Addr_get(hndl, addr, addr_hndl)
lapi_handle_t  hndl;
void         **addr;
int            addr_hndl;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_ADDR_GET(hndl, addr, addr_hndl, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: addr
INTEGER addr_hndl
INTEGER ierror
```

Description

Type of call: local address manipulation

Use this subroutine to get the pointer that was previously registered with LAPI and is associated with the index *addr_hndl*. The value of *addr_hndl* must be in the range 1 <= *addr_hndl* < **LOC_ADDRTBL_SZ**.

Parameters

INPUT

hndl
Specifies the LAPI handle.

addr_hndl
Specifies the index of the function address to retrieve. You should have previously registered the address at this index using `LAPI_Addr_set`. The value of this parameter must be in the range 1 <= *addr_hndl* < **LOC_ADDRTBL_SZ**.

OUTPUT

addr
Returns a function address that the user registered with LAPI.

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To retrieve a header handler address that was previously registered using `LAPI_Addr_set`:

```
lapi_handle_t   hndl;      /* the LAPI handle                       */
void          **addr;      /* the address to retrieve               */
int             addr_hndl; /* the index returned from LAPI_Addr_set */

.

addr_hndl = 1;
LAPI_Addr_get(hndl, &addr, addr_hndl);

/* addr now contains the address that was previously registered */
/* using LAPI_Addr_set                                          */
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_ADDR_HNDL_RANGE
Indicates that the value of *addr_hndl* is not in the range 1 <= *addr_hndl* < **LOC_ADDRTBL_SZ**.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_RET_PTR_NULL
Indicates that the value of the *addr* pointer is NULL (in C) or that the value of *addr* is `LAPI_ADDR_NULL` (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Addr_set Subroutine

Purpose

Registers the address of a function.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Addr_set(hndl, addr, addr_hndl)
lapi_handle_t  hndl;
void          *addr;
int            addr_hndl;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_ADDR_SET(hndl, addr, addr_hndl, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: addr
INTEGER addr_hndl
INTEGER ierror
```

Description

`Type of call:` local address manipulation

Use this subroutine to register the address of a function (*addr*). LAPI maintains the function address in an internal table. The function address is indexed at location *addr_hndl*. In subsequent LAPI calls, *addr_hndl* can be used in place of *addr*. The value of *addr_hndl* must be in the range 1 <= *addr_hndl* < **LOC_ADDRTBL_SZ**.

For active message communication, you can use *addr_hndl* in place of the corresponding header handler address. LAPI only supports this indexed substitution for remote header handler addresses (but not other remote addresses, such as target counters or base data addresses). For these other types of addresses, the actual address value must be passed to the API call.

Parameters

INPUT

hndl
Specifies the LAPI handle.

addr
Specifies the address of the function handler that the user wants to register with LAPI.

addr_hndl
Specifies a user function address that can be passed to LAPI calls in place of a header handler address. The value of this parameter must be in the range 1 <= *addr_hndl* < **LOC_ADDRTBL_SZ**.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To register a header handler address:

```
lapi_handle_t  hndl;      /* the LAPI handle                   */
void          *addr;      /* the remote header handler address */
int            addr_hndl; /* the index to associate            */

⋮

addr = my_func;
addr_hndl = 1;
LAPI_Addr_set(hndl, addr, addr_hndl);

/* addr_hndl can now be used in place of addr in LAPI_Amsend, */
```

```
/* LAPI_Amsendv, and LAPI_Xfer calls                          */
⋮
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_ADDR_HNDL_RANGE
Indicates that the value of *addr_hndl* is not in the range 1 <= *addr_hndl* < **LOC_ADDRTBL_SZ**.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

LAPI_Address Subroutine

Purpose

Returns an unsigned long value for a specified user address.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Address(my_addr, ret_addr)
void   *my_addr;
ulong  *ret_addr;
```

`Note:` This subroutine is meant to be used by FORTRAN programs. The C version of `LAPI_Address` is provided for compatibility purposes only.

FORTRAN Syntax

```
include 'lapif.h'

LAPI_ADDRESS(my_addr, ret_addr, ierror)
INTEGER (KIND=any_type) :: my_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: ret_addr
INTEGER ierror
```

where:

any_type
Is any FORTRAN datatype. This type declaration has the same meaning as the type `void *` in C.

Description

`Type of call:` local address manipulation

Use this subroutine in FORTRAN programs when you need to store specified addresses in an array. In FORTRAN, the concept of address (&) does not exist as it does in C. `LAPI_Address` provides FORTRAN programmers with this function.

Parameters

INPUT

my_addr
Specifies the address to convert. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

ret_addr
Returns the address that is stored in *my_addr* as an unsigned long for use in LAPI calls. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror
Specifies a FORTRAN return code. This is always the last parameter.

FORTRAN Examples

To retrieve the address of a variable:

```
! Contains the address of the target counter
integer (KIND=LAPI_ADDR_TYPE) :: cntr_addr

! Target Counter
type (LAPI_CNTR_T) :: tgt_cntr

! Return code
integer :: ierror

call LAPI_ADDRESS(tgt_cntr, cntr_addr, ierror)

!  cntr_addr now contains the address of tgt_cntr
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_ORG_ADDR_NULL
Indicates that the value of *my_addr* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_ADDR_NULL
Indicates that the value of *ret_addr* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Address_init Subroutine

Purpose

Creates a remote address table.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Address_init(hndl, my_addr, add_tab)
lapi_handle_t  hndl;
```

```
void          *my_addr;
void          *add_tab[ ];
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_ADDRESS_INIT(hndl, my_addr, add_tab, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: my_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: add_tab(*)
INTEGER ierror
```

Description

Type of call: collective communication (blocking)

LAPI_Address_init exchanges virtual addresses among tasks of a parallel application. Use this subroutine to create tables of such items as header handlers, target counters, and data buffer addresses.

LAPI_Address_init is a *collective call* over the LAPI handle *hndl*, which fills the table *add_tab* with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table *add_tab* are passed in using the *my_addr* parameter. Upon completion of this call, *add_tab[i]* contains the virtual address entry that was provided by task *i*. The array is opaque to the user.

Parameters

INPUT

hndl
Specifies the LAPI handle.

my_addr
Specifies the entry supplied by each task. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

add_tab
Specifies the address table containing the addresses that are to be supplied by all tasks. *add_tab* is an array of pointers, the size of which is greater than or equal to **NUM_TASKS**. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To collectively transfer target counter addresses for use in a communication API call, in which all nodes are either 32-bit or 64-bit:

```
lapi_handle_t  hndl;                /* the LAPI handle                    */
void          *addr_tbl[NUM_TASKS]; /* the table for all tasks' addresses */
lapi_cntr_t    tgt_cntr;            /* the target counter                 */

⋮

LAPI_Address_init(hndl, (void *)&tgt_cntr, addr_tbl);

/* for communication with task t, use addr_tbl[t] */
/* as the address of the target counter           */

⋮
```

For a combination of 32-bit and 64-bit nodes, use LAPI_Address_init64.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_COLLECTIVE_PSS
Indicates that a collective call was made while in persistent subsystem (PSS) mode.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_RET_PTR_NULL
Indicates that the value of the *add_tab* pointer is NULL (in C) or that the value of *add_tab* is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Address_init64 Subroutine

Purpose

Creates a 64-bit remote address table.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Address_init64(hndl, my_addr, add_tab)
lapi_handle_t  hndl;
lapi_long_t    my_addr;
lapi_long_t   *add_tab;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_ADDRESS_INIT64(hndl, my_addr, add_tab, ierror)
INTEGER hndl
INTEGER (KIND=LAPI_ADDR_TYPE) :: my_addr
INTEGER (KIND=LAPI_LONG_LONG_TYPE) :: add_tab(*)
INTEGER ierror
```

Description

Type of call: collective communication (blocking)

LAPI_Address_init64 exchanges virtual addresses among a mixture of 32-bit and 64-bit tasks of a parallel application. Use this subroutine to create 64-bit tables of such items as header handlers, target counters, and data buffer addresses.

LAPI_Address_init64 is a *collective call* over the LAPI handle *hndl*, which fills the 64-bit table *add_tab* with the virtual address entries that each task supplies. Collective calls must be made in the same order at all participating tasks.

The addresses that are stored in the table *add_tab* are passed in using the *my_addr* parameter. Upon completion of this call, *add_tab[i]* contains the virtual address entry that was provided by task *i*. The array is opaque to the user.

Parameters

INPUT

hndl
Specifies the LAPI handle.

my_addr
Specifies the address entry that is supplied by each task. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN). To ensure 32-bit/64-bit interoperability, it is passed as a `lapi_long_t` type in C.

OUTPUT

add_tab
Specifies the 64-bit address table that contains the 64-bit values supplied by all tasks. *add_tab* is an array of type `lapi_long_t` (in C) or LAPI_LONG_LONG_TYPE (in FORTRAN). The size of *add_tab* is greater than or equal to **NUM_TASKS**. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To collectively transfer target counter addresses for use in a communication API call with a mixed task environment (any combination of 32-bit and 64-bit):

```
lapi_handle_t hndl;                /* the LAPI handle                     */
lapi_long_t   addr_tbl[NUM_TASKS]; /* the table for all tasks' addresses */
lapi_long_t   tgt_cntr;            /* the target counter                  */

⋮

LAPI_Address_init64(hndl, (lapi_long_t)&tgt_cntr, addr_tbl);

/* For communication with task t, use addr_tbl[t] as the address */
/* of the target counter. For mixed (32-bit and 64-bit) jobs,    */
/* use the LAPI_Xfer subroutine for communication.               */
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_COLLECTIVE_PSS
Indicates that a collective call was made while in persistent subsystem (PSS) mode.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_RET_PTR_NULL
Indicates that the value of the *add_tab* pointer is NULL (in C) or that the value of *add_tab* is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Amsend Subroutine

Purpose

Transfers a user message to a remote task, obtaining the target address on the remote task from a user-specified header handler.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

typedef void (compl_hndlr_t) (hndl, user_info);

lapi_handle_t *hndl;       /* pointer to LAPI context passed in from LAPI_Amsend */
void          *user_info;  /* buffer (user_info) pointer passed in               */
                           /* from header handler (void *(hdr_hndlr_t))          */


typedef void *(hdr_hndlr_t)(hndl, uhdr, uhdr_len, msg_len, comp_h, user_info);

lapi_handle_t   *hndl;       /* pointer to LAPI context passed in from LAPI_Amsend */
void            *uhdr;       /* uhdr passed in from LAPI_Amsend                    */
uint            *uhdr_len;   /* uhdr_len passed in from LAPI_Amsend                */
ulong           *msg_len;    /* udata_len passed in fom LAPI_Amsend                */
compl_hndlr_t  **comp_h;     /* function address of completion handler             */
                             /* (void (compl_hndlr_t)) that needs to be filled     */
                             /* out by this header handler function.               */
void           **user_info;  /* pointer to the parameter to be passed              */
                             /* in to the completion handler                       */


int LAPI_Amsend(hndl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
                tgt_cntr, org_cntr, cmpl_cntr)

lapi_handle_t  hndl;
uint           tgt;
void          *hdr_hdl;
void          *uhdr;
uint           uhdr_len;
void          *udata;
ulong          udata_len;
lapi_cntr_t   *tgt_cntr;
lapi_cntr_t   *org_cntr;
lapi_cntr_t   *cmpl_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'

INTEGER SUBROUTINE COMPL_H (hndl, user_info)
INTEGER hndl
INTEGER user_info

INTEGER FUNCTION HDR_HDL (hndl, uhdr, uhdr_len, msg_len, comp_h, user_info)
INTEGER hndl
INTEGER uhdr
INTEGER uhdr_len
INTEGER (KIND=LAPI_LONG_TYPE) :: msg_len
EXTERNAL INTEGER FUNCTION comp_h
TYPE (LAPI_ADDR_T) :: user_info

LAPI_AMSEND(hndl, tgt, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
            tgt_cntr, org_cntr, cmpl_cntr,  ierror)
INTEGER hndl
INTEGER tgt
EXTERNAL INTEGER FUNCTION hdr_hdl
INTEGER uhdr
INTEGER uhdr_len
```

```
TYPE (LAPI_ADDR_T) :: udata
INTEGER (KIND=LAPI_LONG_TYPE) :: udata_len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

Description

`Type of call:` point-to-point communication (non-blocking)

Use this subroutine to transfer data to a target task, where it is desirable to run a handler on the target task before message delivery begins or after delivery completes. `LAPI_Amsend` allows the user to provide a header handler and optional completion handler. The header handler is used to specify the target buffer address for writing the data, eliminating the need to know the address on the origin task when the subroutine is called.

User data (*uhdr* and *udata*) are sent to the target task. Once these buffers are no longer needed on the origin task, the origin counter is incremented, which indicates the availability of origin buffers for modification. Using the `LAPI_Xfer` call with the `LAPI_AM_XFER` type provides the same type of transfer, with the option of using a send completion handler instead of the origin counter to specify buffer availability.

Upon arrival of the first data packet at the target, the user's header handler is invoked. Note that a header handler must be supplied by the user because it returns the base address of the buffer in which LAPI will write the data sent from the origin task (*udata*). See *RSCT for AIX 5L: LAPI Programming Guide* for an optimization exception to this requirement that a buffer address be supplied to LAPI for single-packet messages.

The header handler also provides additional information to LAPI about the message delivery, such as the completion handler. `LAPI_Amsend` and similar calls (such as `LAPI_Amsendv` and corresponding `LAPI_Xfer` transfers) also allow the user to specify their own message header information, which is available to the header handler. The user may also specify a completion handler parameter from within the header handler. LAPI will pass the information to the completion handler at execution.

Note that the header handler is run inline by the thread running the LAPI dispatcher. For this reason, the header handler must be non-blocking because no other progress on messages will be made until it returns. It is also suggested that execution of the header handler be simple and quick. The completion handler, on the other hand, is normally enqueued for execution by a separate thread. It is possible to request that the completion handler be run inline. See *RSCT for AIX 5L: LAPI Programming Guide* for more information on inline completion handlers.

If a completion handler was not specified (that is, set to `LAPI_ADDR_NULL` in FORTRAN or its pointer set to NULL in C), the arrival of the final packet causes LAPI to increment the target counter on the remote task and send an internal message back to the origin task. The message causes the completion counter (if it is not NULL in C or `LAPI_ADDR_NULL` in FORTRAN) to increment on the origin task.

If a completion handler was specified, the above steps take place after the completion handler returns. To guarantee that the completion handler has executed on the target, you must wait on the completion counter. See *RSCT for AIX 5L: LAPI Programming Guide* for a time-sequence diagram of events in a `LAPI_Amsend` call.

User details

As mentioned above, the user must supply the address of a header handler to be executed on the target upon arrival of the first data packet. The signature of the header handler is as follows:

```
void *hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len, ulong *msg_len,
               compl_hndlr_t **cmpl_hndlr, void **user_info);
```

The value returned by the header handler is interpreted by LAPI as an address for writing the user data (*udata*) that was passed to the `LAPI_Amsend` call. The *uhdr* and *uhdr_len* parameters are passed by LAPI into the header handler and contain the information passed by the user to the corresponding parameters of the `LAPI_Amsend` call.

Use of LAPI_Addr_set

Remote addresses are commonly exchanged by issuing a collective LAPI_Address_init call within a few steps of initializing LAPI. LAPI also provides the LAPI_Addr_set mechanism, whereby users can register one or more header handler addresses in a table, associating an index value with each address. This index can then be passed to LAPI_Amsend instead of an actual address. On the target side, LAPI will use the index to get the header handler address. Note that, if all tasks use the same index for their header handler, the initial collective communication can be avoided. Each task simply registers its own header handler address using the well-known index. Then, on any LAPI_Amsend calls, the reserved index can be passed to the header handler address parameter.

Role of the header handler

The user optionally returns the address of a completion handler function through the *cmpl_hndlr* parameter and a completion handler parameter through the *user_info* parameter. The address passed through the *user_info* parameter can refer to memory containing a datatype defined by the user and then cast to the appropriate type from within the completion handler if desired.

The signature for a user completion handler is as follows:

```
typedef void (compl_hndlr_t)(lapi_handle_t *hndl, void *completion_param);
```

The argument returned by reference through the *user_info* member of the user's header handler will be passed to the *completion_param* argument of the user's completion handler. See the **C Examples** for an example of setting the completion handler and parameter in the header handler.

As mentioned above, the value returned by the header handler must be an address for writing the user data sent from the origin task. There is one exception to this rule. In the case of a single-packet message, LAPI passes the address of the packet in the receive FIFO, allowing the entire message to be consumed within the header handler. In this case, the header handler should return NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) so that LAPI does not copy the message to a target buffer. See *RSCT for AIX 5L: LAPI Programming Guide* for more information (including a sample header handler that uses this method for fast retrieval of a single-packet message).

Passing additional information through lapi_return_info_t

LAPI allows additional information to be passed to and returned from the header handler by passing a pointer to lapi_return_info_t through the msg_len argument. On return from a header handler that is invoked by a call to LAPI_Amsend, the *ret_flags* member of lapi_return_info_t can contain one of these values: LAPI_NORMAL (the default), LAPI_SEND_REPLY (to run the completion handler inline), or LAPI_LOCAL_STATE (no reply is sent). The *dgsp_handle* member of lapi_return_info_t should not be used in conjunction with LAPI_Amsend.

For a complete description of the lapi_return_info_t type, see *RSCT for AIX 5L: LAPI Programming Guide*

Inline execution of completion handlers

Under normal operation, LAPI uses a separate thread for executing user completion handlers. After the final packet arrives, completion handler pointers are placed in a queue to be handled by this thread. For performance reasons, the user may request that a given completion handler be run inline instead of being placed on this queue behind other completion handlers. This mechanism gives users a greater degree of control in prioritizing completion handler execution for performance-critical messages.

LAPI places no restrictions on completion handlers that are run "normally" (that is, by the completion handler thread). Inline completion handlers should be short and should not block, because no progress can be made while the main thread is executing the handler. The user must use caution with inline completion handlers so that LAPI's internal queues do not fill up while waiting for the handler to complete. I/O operations must not be performed with an inline completion handler.

Parameters

INPUT

hndl
Specifies the LAPI handle.

tgt
Specifies the task ID of the target task. The value of this parameter must be in the range `0` <= *tgt* < NUM_TASKS.

hdr_hdl
Specifies the pointer to the remote header handler function to be invoked at the target. The value of this parameter can take an address handle that has already been registered using `LAPI_Addr_set`. The value of this parameter cannot be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

uhdr
Specifies the pointer to the user header data. This data will be passed to the user header handler on the target. If *uhdr_len* is `0`, The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

uhdr_len
Specifies the length of the user's header. The value of this parameter must be a multiple of the processor's word size in the range `0` <= *uhdr_len* <= MAX_UHDR_SZ.

udata
Specifies the pointer to the user data. If *udata_len* is `0`, The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

udata_len
Specifies the length of the user data in bytes. The value of this parameter must be in the range `0` <= *udata_len* <= the value of LAPI constant `LAPI_MAX_MSG_SZ`.

INPUT/OUTPUT

tgt_cntr
Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the target counter is not updated.

org_cntr
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the origin counter is not updated.

cmpl_cntr
Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the completion counter is not updated.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN
Indicates that the value of *udata_len* is greater than the value of LAPI constant `LAPI_MAX_MSG_SZ`.

LAPI_ERR_HDR_HNDLR_NULL
Indicates that the value of the *hdr_hdl* passed in is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_ADDR_NULL
Indicates that the value of the *udata* parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but the value of *udata_len* is greater than 0.

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_UHDR_LEN
Indicates that the *uhdr_len* value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor's doubleword size.

LAPI_ERR_UHDR_NULL
Indicates that the *uhdr* passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but *uhdr_len* is not 0.

C Examples

To send an active message and then wait on the completion counter:

```
/* header handler routine to execute on target task */
void *hdr_hndlr(lapi_handle_t *hndl, void *uhdr, uint *uhdr_len,
               ulong *msg_len, compl_hndlr_t **cmpl_hndlr,
        void **user_info)
{
/* set completion handler pointer and other information */
/* return base address for LAPI to begin its data copy  */
}

{
    lapi_handle_t hndl;                     /* the LAPI handle                     */
    int           task_id;                  /* the LAPI task ID                    */
    int           num_tasks;                /* the total number of tasks           */
    void         *hdr_hndlr_list[NUM_TASKS]; /* the table of remote header handlers */
    int           buddy;                    /* the communication partner           */
    lapi_cntr_t   cmpl_cntr;                /* the completion counter              */
    int           data_buffer[DATA_LEN];    /* the data to transfer                */

    .
    .
    .
    /* retrieve header handler addresses */
    LAPI_Address_init(hndl, (void *)&hdr_hndlr, hdr_hndlr_list);


    /*
    ** up to this point, all instructions have executed on all
    ** tasks. we now begin differentiating tasks.
    */
    if ( sender ) {                  /* origin task */

        /* initialize data buffer, cmpl_cntr, etc.   */
        .
        .
        .
        /* synchronize before starting data transfer */
        LAPI_Gfence(hndl);


        LAPI_Amsend(hndl, buddy, (void *)hdr_hndlr_list[buddy], NULL,
                    0,&(data_buffer[0]),DATA_LEN*(sizeof(int)),
                    NULL, NULL, cmpl_cntr);

        /* Wait on completion counter before continuing. Completion   */
        /* counter will update when message completes at target.      */

    } else {                                            /* receiver */
        .
        .
        .
        /* to match the origin's synchronization before data transfer */
        LAPI_Gfence(hndl);
    }
```

```
    .
    .
    .
}
```

For a complete program listing, see *RSCT for AIX 5L: LAPI Programming Guide.* Sample code illustrating the LAPI_Amsend call can be found in the LAPI sample files. See *RSCT for AIX 5L: LAPI Programming Guide* for more information about the sample programs that are shipped with LAPI.

Location

/usr/lib/liblapi_r.a

LAPI_Amsendv Subroutine

Purpose

Transfers a user vector to a remote task, obtaining the target address on the remote task from a user-specified header handler.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

typedef void (compl_hndlr_t) (hndl, user_info);
lapi_handle_t  *hndl;       /* the LAPI handle passed in from LAPI_Amsendv */
void           *user_info;  /* the buffer (user_info) pointer passed in    */
                            /* from vhdr_hndlr (void *(vhdr_hndlr_t))      */

typedef lapi_vec_t *(vhdr_hndlr_t) (hndl, uhdr, uhdr_len, len_vec, comp_h, uinfo);

lapi_handle_t  *hndl;       /* pointer to the LAPI handle passed in from LAPI_Amsendv */
void           *uhdr;       /* uhdr passed in from LAPI_Amsendv                       */
uint           *uhdr_len;   /* uhdr_len passed in from LAPI_Amsendv                   */
ulong          *len_vec[ ]; /* vector of lengths passed in LAPI_Amsendv               */
compl_hndlr_t **comp_h;     /* function address of completion handler                 */
                            /* (void (compl_hndlr_t)) that needs to be                */
                            /* filled out by this header handler function             */
void          **user_info;  /* pointer to the parameter to be passed                  */
                            /* in to the completion handler                           */

int LAPI_Amsendv(hndl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec,
                 tgt_cntr, org_cntr, cmpl_cntr);

lapi_handle_t   hndl;
uint            tgt;
void           *hdr_hdl;
void           *uhdr;
uint            uhdr_len;
lapi_vec_t     *org_vec;
lapi_cntr_t    *tgt_cntr;
lapi_cntr_t    *org_cntr;
lapi_cntr_t    *cmpl_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'

INTEGER SUBROUTINE COMPL_H (hndl, user_info)
INTEGER hndl
INTEGER user_info(*)

INTEGER FUNCTION VHDR_HDL (hndl, uhdr, uhdr_len, len_vec, comp_h, user_info)
INTEGER hndl
```

```
INTEGER uhdr
INTEGER uhdr_len
INTEGER (KIND=LAPI_LONG_TYPE) :: len_vec
EXTERNAL INTEGER FUNCTION comp_h
TYPE (LAPI_ADDR_T) :: user_info

LAPI_AMSENDV(hndl, tgt, hdr_hdl, uhdr, uhdr_len, org_vec,
             tgt_cntr, org_cntr, cmpl_cntr, ierror)
INTEGER hndl
INTEGER tgt
EXTERNAL INTEGER FUNCTION hdr_hdl
INTEGER uhdr
INTEGER uhdr_len
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

Description

Type of call: point-to-point communication (non-blocking)

LAPI_Amsendv is the vector-based version of the LAPI_Amsend call. You can use it to specify multi-dimensional and non-contiguous descriptions of the data to transfer. Whereas regular LAPI calls allow the specification of a single data buffer address and length, the vector versions allow the specification of a vector of address and length combinations. Additional information is allowed in the data description on the origin task and the target task.

Use this subroutine to transfer a vector of data to a target task, when you want a handler to run on the target task before message delivery begins or after message delivery completes.

To use LAPI_Amsendv, you must provide a header handler, which returns the address of the target vector description that LAPI uses to write the data that is described by the origin vector. The header handler is used to specify the address of the vector description for writing the data, which eliminates the need to know the description on the origin task when the subroutine is called. The header handler is called upon arrival of the first data packet at the target.

Optionally, you can also provide a completion handler. The header handler provides additional information to LAPI about the message delivery, such as the completion handler. You can also specify a completion handler parameter from within the header handler. LAPI passes the information to the completion handler at execution.

With the exception of the address that is returned by the completion handler, the use of counters, header handlers, and completion handlers in LAPI_Amsendv is identical to that of LAPI_Amsend. In both cases, the user header handler returns information that LAPI uses for writing at the target. See LAPI_Amsend for more information. This section presents information that is specific to the vector version of the call (LAPI_Amsendv).

LAPI vectors are structures of type lapi_vec_t, defined as follows:

```
typedef struct  {
    lapi_vectype_t  vec_type;
    uint            num_vecs;
    void            **info;
    ulong           *len;
} lapi_vec_t;
```

vec_type is an enumeration that describes the type of vector transfer, which can be: LAPI_GEN_GENERIC, LAPI_GEN_IOVECTOR, or LAPI_GEN_STRIDED_XFER.

For transfers of type LAPI_GEN_GENERIC and LAPI_GEN_IOVECTOR, the fields are used as follows:

num_vecs
indicates the number of data vectors to transfer. Each data vector is defined by a base address and data length.

info
is the array of addresses.

len
is the array of data lengths.

For example, consider the following vector description:

```
vec_type = LAPI_GEN_IOVECTOR
num_vecs = 3
info     = {addr_0, addr_1, addr_2}
len      = {len_0, len_1, len_2}
```

On the origin side, this example would tell LAPI to read len_0 bytes from addr_0, len_1 bytes from addr_1, and len_2 bytes from addr_2. As a target vector, this example would tell LAPI to write len_0 bytes to addr_0, len_1 bytes to addr_1, and len_2 bytes to addr_2.

Recall that vector transfers require an origin and target vector. For LAPI_Amsendv calls, the origin vector is passed to the API call on the origin task. The address of the target vector is returned by the header handler.

For transfers of type LAPI_GEN_GENERIC, the target vector description must also have type LAPI_GEN_GENERIC. The contents of the *info* and *len* arrays are unrestricted in the generic case; the number of vectors and the length of vectors on the origin and target do not need to match. In this case, LAPI transfers a given number of bytes in noncontiguous buffers specified by the origin vector to a set of noncontiguous buffers specified by the target vector.

If the sum of target vector data lengths (say TGT_LEN) is less than the sum of origin vector data lengths (say ORG_LEN), only the first TGT_LEN bytes from the origin buffers are transferred and the remaining bytes are discarded. If TGT_LEN is greater than ORG_LEN, all ORG_LEN bytes are transferred. Consider the following example:

```
Origin_vector: {
    num_vecs = 3;
    info     = {orgaddr_0, orgaddr_1, orgaddr_2};
    len      = {5, 10, 5}
}

Target_vector: {
    num_vecs = 4;
    info     = {tgtaddr_0, tgtaddr_1, tgtaddr_2, tgtaddr_3};
    len      = {12, 2, 4, 2}
}
```

LAPI copies data as follows:

1. 5 bytes from orgaddr_0 to tgtaddr_0 (leaves 7 bytes of space at a 5-byte offset from tgtaddr_0)
2. 7 bytes from orgaddr_1 to remaining space in tgtaddr_0 (leaves 3 bytes of data to transfer from orgaddr_1)
3. 2 bytes from orgaddr_1 to tgtaddr_1 (leaves 1 byte to transfer from orgaddr_1)
4. 1 byte from orgaddr_1 followed by 3 bytes from orgaddr_2 to tgt_addr_2 (leaves 3 bytes to transfer from orgaddr_2)
5. 2 bytes from orgaddr_2 to tgtaddr_3

LAPI will copy data from the origin until the space described by the target is filled. For example:

```
Origin_vector: {
    num_vecs = 1;
    info     = {orgaddr_0};
    len      = {20}
}

Target_vector: {
    num_vecs = 2;
    info     = {tgtaddr_0, tgtaddr_1};
    len      = {5, 10}
}
```

LAPI will copy 5 bytes from orgaddr_0 to tgtaddr_0 and the next 10 bytes from orgaddr_0 to tgtaddr_1. The remaining 5 bytes from orgaddr_0 will not be copied.

For transfers of type LAPI_GEN_IOVECTOR, the lengths of the vectors must match and the target vector description must match the origin vector description. More specifically, the target vector description must:

- also have type LAPI_GEN_IOVECTOR
- have the same *num_vecs* as the origin vector
- initialize the info array with *num_vecs* addresses in the target address space. For LAPI vectors origin_vector and target_vector described similarly to the example above, data is copied as follows:
  1. transfer origin_vector.len[0] bytes from the address at origin_vector.info[0] to the address at target_vector.info[0]
  2. transfer origin_vector.len[1] bytes from the address at origin_vector.info[1] to the address at target_vector.info[1]
  3. transfer origin_vector.len[*n*] bytes from the address at origin_vector.info[*n*] to the address at target_vector.info[*n*], for *n* = 2 to *n* = [*num_vecs*-3]
  4. transfer origin_vector.len[*num_vecs*-2] bytes from the address at origin_vector.info[*num_vecs*-2] to the address at target_vector.info[*num_vecs*-2]
  5. copy origin_vector.len[*num_vecs*-1] bytes from the address at origin_vector.info[*num_vecs*-1] to the address at target_vector.info[*num_vecs*-1]

Strided vector transfers

For transfers of type LAPI_GEN_STRIDED_XFER, the target vector description must match the origin vector description. Rather than specifying the set of address and length pairs, the *info* array of the origin and target vectors is used to specify a data block "template", consisting of a base address, block size and stride. LAPI thus expects the *info* array to contain three integers. The first integer contains the base address, the second integer contains the block size to copy, and the third integer contains the byte stride. In this case, *num_vecs* indicates the number of blocks of data that LAPI should copy, where the first block begins at the base address. The number of bytes to copy in each block is given by the block size and the starting address for all but the first block is given by previous address + stride. The total amount of data to be copied will be *num_vecs***block_size*. Consider the following example:

```
Origin_vector {
    num_vecs = 3;
    info     = {orgaddr, 5, 8}
}
```

Based on this description, LAPI will transfer 5 bytes from orgaddr, 5 bytes from orgaddr+8 and 5 bytes from orgaddr+16.

Call details

As mentioned above, counter and handler behavior in LAPI_Amsendv is nearly identical to that of LAPI_Amsend. A short summary of that behavior is provided here. See the LAPI_Amsend description for full details.

This is a non-blocking call. The calling task cannot change the *uhdr* (origin header) and *org_vec* data until completion at the origin is signaled by the *org_cntr* being incremented. The calling task cannot assume that the *org_vec* structure can be changed before the origin counter is incremented. The structure (of type lapi_vec_t) that is returned by the header handler cannot be modified before the target counter has been incremented. Also, if a completion handler is specified, it may execute asynchronously, and can only be assumed to have completed after the target counter increments (on the target) or the completion counter increments (at the origin).

The length of the user-specified header (*uhdr_len*) is constrained by the implementation-specified maximum value **MAX_UHDR_SZ**. *uhdr_len* must be a multiple of the processor's doubleword size. To get the best bandwidth, *uhdr_len* should be as small as possible.

If the following requirement is not met, an error condition occurs:

- If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

Parameters

hndl
Specifies the LAPI handle.

tgt
Specifies the task ID of the target task. The value of this parameter must be in the range `0` <= *tgt* < **NUM_TASKS**.

hdr_hdl
Points to the remote header handler function to be invoked at the target. The value of this parameter can take an address handle that had been previously registered using the `LAPI_Addr_set/LAPI_Addr_get` mechanism. The value of this parameter cannot be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

uhdr
Specifies the pointer to the local header (parameter list) that is passed to the handler function. If *uhdr_len* is `0`, The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

uhdr_len
Specifies the length of the user's header. The value of this parameter must be a multiple of the processor's doubleword size in the range `0` <= *uhdr_len* <= **MAX_UHDR_SZ**.

org_vec
Points to the origin vector.

INPUT/OUTPUT

tgt_cntr
Specifies the target counter address. The target counter is incremented after the completion handler (if specified) completes or after the completion of data transfer. If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the target counter is not updated.

org_cntr
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data is copied out of the origin address (in C) or the origin (in FORTRAN). If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the origin counter is not updated.

cmpl_cntr
Specifies the counter at the origin that signifies completion of the completion handler. It is updated once the completion handler completes. If no completion handler is specified, the counter is incremented at the completion of message delivery. If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the completion counter is not updated.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

1. To send a `LAPI_GEN_IOVECTOR` using active messages:

```
/* header handler routine to execute on target task */
lapi_vec_t *hdr_hndlr(lapi_handle_t *handle, void *uhdr, uint *uhdr_len,
                      ulong *len_vec[ ], compl_hndlr_t **completion_handler,
              void **user_info)
{
     /* set completion handler pointer and other info */
```

```
     /* set up the vector to return to LAPI                                 */
     /* for a LAPI_GEN_IOVECTOR: num_vecs, vec_type, and len must all have */
     /* the same values as the origin vector.  The info array should       */
     /* contain the buffer addresses for LAPI to write the data            */
     vec->num_vecs = NUM_VECS;
     vec->vec_type = LAPI_GEN_IOVECTOR;
     vec->len      = (unsigned long *)malloc(NUM_VECS*sizeof(unsigned long));
     vec->info     = (void **) malloc(NUM_VECS*sizeof(void *));
     for( i=0; i < NUM_VECS; i++ ) {
         vec->info[i] = (void *) &data_buffer[i];
         vec->len[i]  = (unsigned long)(sizeof(int));
     }

     return vec;
}

{


     .
     .
     .

     void        *hdr_hndlr_list[NUM_TASKS]; /* table of remote header handlers */
     lapi_vec_t  *vec;                       /* data for data transfer          */


     vec->num_vecs = NUM_VECS;
     vec->vec_type = LAPI_GEN_IOVECTOR;
     vec->len      = (unsigned long *) malloc(NUM_VECS*sizeof(unsigned long));
     vec->info     = (void **) malloc(NUM_VECS*sizeof(void *));

     /* each vec->info[i] gets a base address                                 */
     /* each vec->len[i] gets the number of bytes to transfer from vec->info[i] */

     LAPI_Amsendv(hndl, tgt, (void *) hdr_hdl_list[buddy], NULL, 0, vec,
                  tgt_cntr, org_cntr, cmpl_cntr);

     /* data will be copied as follows:                    */
     /* len[0] bytes of data starting from address info[0] */
     /* len[1] bytes of data starting from address info[1] */
     .
     .
     .
     /* len[NUM_VECS-1] bytes of data starting from address info[NUM_VECS-1] */

}
```

The above example could also illustrate the LAPI_GEN_GENERIC type, with the following modifications:

- Both vectors would need LAPI_GEN_GENERIC as the vec_type.
- There are no restrictions on symmetry of number of vectors and lengths between the origin and target sides.

2. To send a LAPI_STRIDED_VECTOR using active messages:

```
/* header handler routine to execute on target task */
lapi_vec_t *hdr_hndlr(lapi_handle_t *handle, void *uhdr, uint *uhdr_len,
                      ulong *len_vec[ ], compl_hndlr_t **completion_handler,
              void **user_info)
{

     int block_size;            /* block size */
     int data_size;             /* stride     */
     .
     .
     .
     vec->num_vecs = NUM_VECS;     /* NUM_VECS = number of vectors to transfer */
                                   /* must match that of the origin vector     */
     vec->vec_type = LAPI_GEN_STRIDED_XFER;           /* same as origin vector */

     /* see comments in origin vector setup for a description of how data      */
     /* will be copied based on these settings.                                */
     vec->info[0]  = buffer_address; /* starting address for data copy         */
     vec->info[1]  = block_size;     /* bytes of data to copy                  */
     vec->info[2]  = stride;         /* distance from copy block to copy block */
     .
```

```
     .
     .
     return vec;

}

{
      .
      .
      .
      lapi_vec_t *vec;                                  /* data for data transfer */

      vec->num_vecs = NUM_VECS;        /* NUM_VECS = number of vectors to transfer */
                                       /* must match that of the target vector     */
      vec->vec_type = LAPI_GEN_STRIDED_XFER;          /* same as target vector     */

      vec->info[0]  = buffer_address;    /* starting address for data copy         */
      vec->info[1]  = block_size;        /* bytes of data to copy                  */
      vec->info[2]  = stride;            /* distance from copy block to copy block */
      /* data will be copied as follows:                                           */
      /* block_size bytes will be copied from buffer_address                       */
      /* block_size bytes will be copied from buffer_address+stride                */
      /* block_size bytes will be copied from buffer_address+(2*stride)            */
      /* block_size bytes will be copied from buffer_address+(3*stride)            */
      .
      .
      .
      /* block_size bytes will be copied from buffer_address+((NUM_VECS-1)*stride) */
      .
      .
      .
      /* if uhdr isn't used, uhdr should be NULL and uhdr_len should be 0          */
      /* tgt_cntr, org_cntr and cmpl_cntr can all be NULL                          */
      LAPI_Amsendv(hndl, tgt, (void *) hdr_hdl_list[buddy], uhdr, uhdr_len,
                   vec, tgt_cntr, org_cntr, cmpl_cntr);
     .
     .
     .

}
```

For complete examples, see the sample programs shipped with LAPI.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HDR_HNDLR_NULL
Indicates that the *hdr_hdl* passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_EXTENT
Indicates that the *org_vec*'s extent (stride * *num_vecs*) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE
Indicates that the *org_vec* stride is less than block.

LAPI_ERR_ORG_VEC_ADDR
Indicates that the *org_vec->info[i]* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (*org_vec->len[i]*) is not 0.

LAPI_ERR_ORG_VEC_LEN
Indicates that the sum of *org_vec->len* is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_VEC_NULL
Indicates that *org_vec* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_ORG_VEC_TYPE
Indicates that the *org_vec->vec_type* is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL
Indicates that the strided vector address *org_vec*->*info*[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_UHDR_LEN
Indicates that the *uhdr_len* value passed in is greater than **MAX_UHDR_SZ** or is not a multiple of the processor's doubleword size.

LAPI_ERR_UHDR_NULL
Indicates that the *uhdr* passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but *uhdr_len* is not 0.

Location

/usr/lib/liblapi_r.a

LAPI_Fence Subroutine

Purpose

Enforces order on LAPI calls.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Fence(hndl)
lapi_handle_t hndl;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_FENCE(hndl, ierror)
INTEGER hndl
INTEGER ierror
```

Description

Type of call: Local data synchronization (blocking) (may require progress on the remote task)

Use this subroutine to enforce order on LAPI calls. If a task calls LAPI_Fence, all the LAPI operations that were initiated by that task, before the fence using the LAPI context *hndl*, are guaranteed to complete at the target tasks. This occurs before any of its communication operations using *hndl*, initiated after the LAPI_Fence, start transmission of data. This is a data fence which means that the data movement is complete. This is not an operation fence which would need to include active message completion handlers completing on the target.

LAPI_Fence may require internal protocol processing on the remote side to complete the fence request.

Parameters

INPUT

hndl
Specifies the LAPI handle.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

C Examples

To establish a data barrier in a single task:

```
lapi_handle_t hndl;  /* the LAPI handle */

⋮

/* API communication call 1 */
/* API communication call 2 */

⋮

/* API communication call n */

LAPI_Fence(hndl);

/* all data movement from above communication calls has completed by this point */
/* any completion handlers from active message calls could still be running.    */
```

Location

/usr/lib/liblapi_r.a

LAPI_Get Subroutine

Purpose

Copies data from a remote task to a local task.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Get(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr)
lapi_handle_t hndl;
uint          tgt;
ulong         len;
void         *tgt_addr;
void         *org_addr;
lapi_cntr_t  *tgt_cntr;
lapi_cntr_t  *org_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_GET(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_LONG_TYPE) :: len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: org_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

Description

`Type of call:` point-to-point communication (non-blocking)

Use this subroutine to transfer data from a remote (target) task to a local (origin) task. Note that in this case the origin task is actually the *receiver* of the data. This difference in transfer type makes the counter behavior slightly different than in the normal case of origin sending to target.

The origin buffer will still increment on the origin task upon availability of the origin buffer. But in this case, the origin buffer becomes available once the transfer of data is complete. Similarly, the target counter will increment once the target buffer is available. Target buffer availability in this case refers to LAPI no longer needing to access the data in the buffer.

This is a non-blocking call. The caller *cannot* assume that data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer addresses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Parameters

INPUT

hndl
Specifies the LAPI handle.

tgt
Specifies the task ID of the target task that is the source of the data. The value of this parameter must be in the range `0` <= *tgt* < **NUM_TASKS**.

len
Specifies the number of bytes of data to be copied. This parameter must be in the range `0` <= *len* <= the value of LAPI constant `LAPI_MAX_MSG_SZ`.

tgt_addr
Specifies the target buffer address of the data source. If *len* is `0`, The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

INPUT/OUTPUT

tgt_cntr
Specifies the target counter address. The target counter is incremented once the data buffer on the target can be modified. If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the target counter is not updated.

org_cntr
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data arrives at the origin. If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the origin counter is not updated.

OUTPUT

org_addr
Specifies the local buffer address into which the received data is copied. If *len* is 0, The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

```
{
    /* initialize the table buffer for the data addresses */

    /* get remote data buffer addresses                   */
    LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
    .
    .
    .
    LAPI_Get(hndl, tgt, (ulong) data_len, (void *) (data_buffer_list[tgt]),
             (void *) data_buffer, tgt_cntr, org_cntr);

    /* retrieve data_len bytes from address data_buffer_list[tgt] on task tgt. */
    /* write the data starting at address data_buffer. tgt_cntr and org_cntr   */
    /* can be NULL. */
}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN
Indicates that the value of *udata_len* is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_ADDR_NULL
Indicates that the *org_addr* passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but *len* is greater than 0.

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_ADDR_NULL
Indicates that the *tgt_addr* passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but *len* is greater than 0.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

Location

/usr/lib/liblapi_r.a

LAPI_Getcntr Subroutine

Purpose

Gets the integer value of a specified LAPI counter.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Getcntr(hndl, cntr, val)
lapi_handle_t  hndl;
lapi_cntr_t   *cntr;
int           *val;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_GETCNTR(hndl, cntr, val,  ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER ierror
```

Description

`Type of call:` Local counter manipulation

This subroutine gets the integer value of *cntr*. It is used to check progress on *hndl*.

Parameters

INPUT

hndl
Specifies the LAPI handle.

cntr
Specifies the address of the counter. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

val
Returns the integer value of the counter *cntr*. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

```
{
    lapi_cntr_t cntr;
    int         val;

   /* cntr is initialized */

   /* processing/communication takes place */

   LAPI_Getcntr(hndl, &cntr, &val)

   /* val now contains the current value of cntr */
}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL
Indicates that the *cntr* pointer is NULL (in C) or that the value of *cntr* is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_RET_PTR_NULL
Indicates that the value of the *val* pointer is NULL (in C) or that the value of *val* is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Getv Subroutine

Purpose

Copies vectors of data from a remote task to a local task.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr)
lapi_handle_t hndl;
uint          tgt;
lapi_vec_t   *tgt_vec;
lapi_vec_t   *org_vec;
lapi_cntr_t  *tgt_cntr;
lapi_cntr_t  *org_cntr;


typedef struct  {
    lapi_vectype_t  vec_type;  /* operation code        */
    uint            num_vecs;  /* number of vectors     */
    void            **info;    /* vector of information */
    ulong           *len;      /* vector of lengths     */
} lapi_vec_t;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_GETV(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_vec
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

The 32-bit version of the LAPI_VEC_T type is defined as:

```
TYPE LAPI_VEC_T
   SEQUENCE
   INTEGER(KIND = 4)  :: vec_type
   INTEGER(KIND = 4)  :: num_vecs
   INTEGER(KIND = 4)  :: info
   INTEGER(KIND = 4)  :: len
END TYPE LAPI_VEC_T
```

The 64-bit version of the LAPI_VEC_T type is defined as:

```
TYPE LAPI_VEC_T
   SEQUENCE
   INTEGER(KIND = 4)  :: vec_type
   INTEGER(KIND = 4)  :: num_vecs
   INTEGER(KIND = 8)  :: info
   INTEGER(KIND = 8)  :: len
END TYPE LAPI_VEC_T
```

Description

Type of call: point-to-point communication (non-blocking)

This subroutine is the vector version of the LAPI_Get call. Use LAPI_Getv to transfer vectors of data from the target task to the origin task. Both the origin and target vector descriptions are located in the address space of the origin task. But, the values specified in the *info* array of the target vector must be addresses in the address space of the target task.

The calling program *cannot* assume that the origin buffer can be changed or that the contents of the origin buffers on the origin task are ready for use upon function return. After the origin counter (*org_cntr*) is incremented, the origin buffers can be modified by the origin task. After the target counter (*tgt_cntr*) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (*cmpl_cntr*), it is incremented at the origin after the target counter (*tgt_cntr*) has been incremented at the target. If the values of any of the counters or counter addresses are NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If any of the following requirements are not met, an error condition occurs:

- The vector types *org_vec->vec_type* and *tgt_vec->vec_type* must be the same.
- If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.
- The length of any vector that is pointed to by *tgt_vec* must be equal to the length of the corresponding vector that is pointed to by *org_vec*.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the origin side, the contents of the origin buffer are undefined after the operation.

See LAPI_Amsendv for details about commuication using different LAPI vector types. (LAPI_Getv does not support the LAPI_GEN_GENERIC type.)

Parameters

INPUT

hndl
Specifies the LAPI handle.

tgt
Specifies the task ID of the target task. The value of this parameter must be in the range 0 <= *tgt* < **NUM_TASKS**.

tgt_vec
Points to the target vector description.

org_vec
Points to the origin vector description.

INPUT/OUTPUT

tgt_cntr
Specifies the target counter address. The target counter is incremented once the data buffer on the target can be modified. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

org_cntr
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented after data arrives at the origin. If the value of this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To get a LAPI_GEN_IOVECTOR:

```
{

      /* retrieve a remote data buffer address for data to transfer,  */
      /* such as through LAPI_Address_init                            */

      /* task that calls LAPI_Getv sets up both org_vec and tgt_vec   */
      org_vec->num_vecs = NUM_VECS;
      org_vec->vec_type = LAPI_GEN_IOVECTOR;
      org_vec->len = (unsigned long *)
      malloc(NUM_VECS*sizeof(unsigned long));
      org_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));

      /* each org_vec->info[i] gets a base address on the origin task */
      /* each org_vec->len[i] gets the number of bytes to write to    */
      /* org_vec->info[i]                                             */

      tgt_vec->num_vecs = NUM_VECS;
      tgt_vec->vec_type = LAPI_GEN_IOVECTOR;
      tgt_vec->len = (unsigned long *)
      malloc(NUM_VECS*sizeof(unsigned long));
      tgt_vec->info = (void **) malloc(NUM_VECS*sizeof(void *));

      /* each tgt_vec->info[i] gets a base address on the target task */
      /* each tgt_vec->len[i] gets the number of bytes to transfer    */
      /* from vec->info[i]                                            */
      /* For LAPI_GEN_IOVECTOR, num_vecs, vec_type, and len must be   */
      /* the same                                                     */

      LAPI_Getv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr);
      /* tgt_cntr and org_cntr can both be NULL           */

      /* data will be retrieved as follows:               */
      /* org_vec->len[0] bytes will be retrieved from     */
      /* tgt_vec->info[0] and written to org_vec->info[0] */
      /* org_vec->len[1] bytes will be retrieved from     */
      /* tgt_vec->info[1] and written to org_vec->info[1] */
      .
      .
      .
      /* org_vec->len[NUM_VECS-1] bytes will be retrieved */
      /* from tgt_vec->info[NUM_VECS-1] and written to    */
      /* org_vec->info[NUM_VECS-1]                        */

}
```

For examples of other vector types, see LAPI_Amsendv.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_EXTENT
Indicates that the *org_vec*'s extent (stride * *num_vecs*) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE
Indicates that the *org_vec* stride is less than block size.

LAPI_ERR_ORG_VEC_ADDR
Indicates that some *org_vec*->*info*[*i*] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN). but the corresponding length (*org_vec*->*len*[*i*]) is not 0.

LAPI_ERR_ORG_VEC_LEN
Indicates that the total sum of all *org_vec*->*len*[*i*] (where [*i*] is in the range 0 <= *i* <= *org_vec*->*num_vecs*) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_VEC_NULL
Indicates that the *org_vec* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_ORG_VEC_TYPE
Indicates that the *org_vec*->*vec_type* is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL
Indicates that the strided vector base address *org_vec*->*info*[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL
Indicates that the strided vector address *tgt_vec*->*info*[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_EXTENT
Indicates that *tgt_vec*'s extent (stride * *num_vecs*) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_TGT_STRIDE
Indicates that the *tgt_vec*'s stride is less than its block size.

LAPI_ERR_TGT_VEC_ADDR
Indicates that the *tgt_vec*->*info*[i] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (*tgt_vec*->*len*[i]) is not 0.

LAPI_ERR_TGT_VEC_LEN
Indicates that the sum of *tgt_vec*->*len* is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_VEC_NULL
Indicates that *tgt_vec* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_VEC_TYPE
Indicates that the *tgt_vec*->*vec_type* is not valid.

LAPI_ERR_VEC_LEN_DIFF
Indicates that *org_vec* and *tgt_vec* have different lengths (*len*[]).

LAPI_ERR_VEC_NUM_DIFF
Indicates that *org_vec* and *tgt_vec* have different *num_vecs*.

LAPI_ERR_VEC_TYPE_DIFF
Indicates that *org_vec* and *tgt_vec* have different vector types (*vec_type*).

Location

/usr/lib/liblapi_r.a

LAPI_Gfence Subroutine

Purpose

Enforces order on LAPI calls across all tasks and provides barrier synchronization among them.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Gfence(hndl)
lapi_handle_t hndl;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_GFENCE(hndl, ierror)
INTEGER hndl
INTEGER ierror
```

Description

`Type of call:` collective data synchronization (blocking)

Use this subroutine to enforce global order on LAPI calls. This is a *collective call.* Collective calls must be made in the same order at all participating tasks.

On completion of this call, it is assumed that all LAPI communication associated with *hndl* from all tasks has quiesced. Although *hndl* is local, it represents a set of tasks that were associated with it at `LAPI_Init`, all of which must participate in this operation for it to complete. This is a data fence, which means that the data movement is complete. This is not an operation fence, which would need to include active message completion handlers completing on the target.

Parameters

INPUT

hndl
Specifies the LAPI handle.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

LAPI_Init Subroutine

Purpose

Initializes a LAPI context.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Init(hndl,lapi_info)
lapi_handle_t *hndl;
lapi_info_t *lapi_info;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_INIT(hndl,lapi_info,ierror)
INTEGER hndl
TYPE (LAPI_INFO_T) :: lapi_info
INTEGER ierror
```

Description

`Type of call:` Local initialization

Use this subroutine to instantiate and initialize a new LAPI context. A handle to the newly-created LAPI context is returned in *hndl*. All subsequent LAPI calls can use *hndl* to specify the context of the LAPI operation. Except for `LAPI_Address()` and `LAPI_Msg_string()`, the user cannot make any LAPI calls before calling `LAPI_Init()`.

The *lapi_info* structure (`lapi_info_t`) must be "zeroed out" before any fields are filled in. To do this in C, use this statement: `bzero (lapi_info, size of (lapi_info_t))`. In FORTRAN, you need to "zero out" each field manually in the LAPI_INFO_T type. Fields with a description of `Future support` should not be used because the names of those fields might change.

The `lapi_info_t` structure is defined as follows:

```
typedef struct {
    lapi_dev_t     protocol;      /* Protocol device returned                    */
    lapi_lib_t     lib_vers;      /* LAPI library version -- user-supplied       */
    uint           epoch_num;     /* No longer used                              */
    int            num_compl_hndlr_thr; /* Number of completion handler threads  */
    uint           instance_no;   /* Instance of LAPI to initialize [1-16]       */
    int            info6;         /* Future support                              */
    LAPI_err_hndlr *err_hndlr;    /* User-registered error handler               */
    com_thread_info_t *lapi_thread_attr; /* Support thread att and init function */
    void           *adapter_name; /* What adapter to initialize, i.e. css0, ml0  */
    lapi_extend_t  *add_info;     /* Additional structure extension              */
} lapi_info_t;
```

The fields are used as follows:

protocol
LAPI sets this field to the protocol that has been initialized.

lib_vers
Is used to indicate a library version to LAPI for compatibility purposes. Valid values for this field are:

L1_LIB
Provides basic functionality (this is the default).

L2_LIB
Provides the ability to use counters as structures.

LAST_LIB
Provides the most current level of functionality. For new users of LAPI, *lib_vers* should be set to LAST_LIB.

This field must be set to L2_LIB or LAST_LIB to use LAPI_Nopoll_wait and LAPI_Setcntr_wstatus.

epoch_num
This field is no longer used.

num_compl_hndlr_thr
Indicates to LAPI the number of completion handler threads to initialize.

instance_no
Specifies the instance of LAPI to initialize (1 to 16).

info6
This field is for future use.

err_hndlr
Use this field to optionally pass a callback pointer to an error-handler routine.

lapi_thread_attr
Supports thread attributes and initialization function.

adapter_name
Is used in persistent subsystem (PSS) mode to pass an adapter name.

add_info
Is used for additional information in standalone UDP mode.

Parameters

INPUT/OUTPUT

lapi_info
Specifies a structure that provides the parallel job information with which this LAPI context is associated. The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

OUTPUT

hndl
Specifies a pointer to the LAPI handle to initialize.

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_ALL_HNDL_IN_USE
All available LAPI instances are in use.

LAPI_ERR_BOTH_NETSTR_SET
Both the MP_LAPI_NETWORK and MP_LAPI_INET statements are set (only one should be set).

LAPI_ERR_CSS_LOAD_FAILED
LAPI is unable to load the communication utility library.

LAPI_ERR_HNDL_INVALID
The lapi_handle_t * passed to LAPI for initialization is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_INFO_NONZERO_INFO
The future support fields in the `lapi_info_t` structure that was passed to LAPI are not set to zero (and should be).

LAPI_ERR_INFO_NULL
The `lapi_info_t` pointer passed to LAPI is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

LAPI_ERR_MEMORY_EXHAUSTED
LAPI is unable to obtain memory from the system.

LAPI_ERR_MSG_API
Indicates that the `MP_MSG_API` environment variable is not set correctly.

LAPI_ERR_NO_NETSTR_SET
No network statement is set. Note that if running with POE, this will be returned if `MP_MSG_API` is not set correctly.

LAPI_ERR_NO_UDP_HNDLR
You passed a value of NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN) for both the UDP handler and the UDP list. One of these (the UDP handler or the UDP list) must be initialized for standalone UDP initialization. This error is returned in standalone UDP mode only.

LAPI_ERR_PSS_NON_ROOT
You tried to initialize the persistent subsystem (PSS) protocol as a nonroot user.

LAPI_ERR_SHM_KE_NOT_LOADED
LAPI's shared memory kernel extension is not loaded.

LAPI_ERR_SHM_SETUP
LAPI is unable to set up shared memory. This error will be returned if `LAPI_USE_SHM=only` and tasks are assigned to more than one node.

LAPI_ERR_UDP_PKT_SZ
The UDP packet size you indicated is not valid.

LAPI_ERR_UNKNOWN
An internal error has occurred.

LAPI_ERR_USER_UDP_HNDLR_FAIL
The UDP handler you passed has returned a non-zero error code. This error is returned in standalone UDP mode only.

C Examples

The following environment variable must be set before LAPI is initialized:

```
MP_MSG_API=[ lapi | [ lapi,mpi | mpi,lapi ] | mpi_lapi ]
```

The following environment variables are also commonly used:

```
MP_EUILIB=[ ip | us ] (ip is the default)
MP_PROCS=number_of_tasks_in_job
LAPI_USE_SHM=[ yes | no | only ] (no is the default)
```

To initialize LAPI, follow these steps:

1. Set environment variables (as described in *RSCT for AIX 5L: LAPI Programming Guide*) before the user application is invoked. The remaining steps are done in the user application.
2. Clear `lapi_info_t`, then set any fields.
3. Call `LAPI_Init`.

For systems running PE

Both US and UDP/IP are supported for shared handles as long as they are the same for both handles. Mixed transport protocols such as LAPI IP and shared user space (US) are not supported.

To initialize a LAPI handle:

```
{
    lapi_handle_t hndl;
    lapi_info_t info;

    bzero(&info, sizeof(lapi_info_t));  /* clear lapi_info */

    LAPI_Init(&hndl, &info);
}
```

To initialize a LAPI handle and register an error handler:

```
void my_err_hndlr(lapi_handle_t *hndl, int *error_code, lapi_err_t *err_type,
                  int *task_id, int *src )
{
   /* examine passed parameters and delete desired information */

   if ( user wants to terminate ) {
       LAPI_Term(*hndl);               /* will terminate LAPI */
       exit(some_return_code);
   }

   /* any additional processing */

   return; /* signals to LAPI that error is non-fatal; execution should continue */
}

{
    lapi_handle_t hndl;
    lapi_info_t info;


    bzero(&info, sizeof(lapi_info_t));  /* clear lapi_info */

    /* set error handler pointer */
    info.err_hndlr = (LAPI_err_hndlr) my_err_hndlr;

    LAPI_Init(&hndl, &info);
}
```

For standalone systems (not running PE)

To initialize a LAPI handle for UDP/IP communication using a user handler:

```
int my_udp_hndlr(lapi_handle_t *hndl, lapi_udp_t *local_addr, lapi_udp_t *addr_list,
                lapi_udpinfo_t *info)

{
    /* LAPI will allocate and free addr_list pointer when using */
    /* a user handler                                           */

    /* use the AIX(r) inet_addr call to convert an IP address      */
    /* from a dotted quad to a long                             */
    task_0_ip_as_long = inet_addr(task_0_ip_as_string);
    addr_list[0].ip_addr = task_0_ip_as_long;
    addr_list[0].port_no = task_0_port_as_unsigned;

    task_1_ip_as_long = inet_addr(task_1_ip_as_string);
    addr_list[1].ip_addr = task_1_ip_as_long;
    addr_list[1].port_no = task_1_port_as_unsigned;
    .
    .
    .
    task_num_tasks-1_ip_as_long = inet_addr(task_num_tasks-1_ip_as_string);
    addr_list[num_tasks-1].ip_addr = task_num_tasks-1_ip_as_long;
    addr_list[num_tasks-1].port_no = task_num_tasks-1_port_as_unsigned;

}


{
```

```
    lapi_handle_t hndl;
    lapi_info_t info;
    lapi_extend_t extend_info;

    bzero(&info, sizeof(lapi_info_t));           /* clear lapi_info        */
    bzero(&extend_info, sizeof(lapi_extend_t));  /* clear lapi_extend_info */

    extend_info.udp_hndlr = (udp_init_hndlr *) my_udp_hndlr;
    info.add_info = &extend_info;


    LAPI_Init(&hndl, &info);
}
```

To initialize a LAPI handle for UDP/IP communication using a user list:

```
{
    lapi_handle_t hndl;
    lapi_info_t info;
    lapi_extend_t extend_info;
    lapi_udp_t *addr_list;

    bzero(&info, sizeof(lapi_info_t));           /* clear lapi_info        */
    bzero(&extend_info, sizeof(lapi_extend_t));  /* clear lapi_extend_info */

    /* when using a user list, the user is responsible for allocating          */
    /* and freeing the list pointer                                            */
    addr_list = malloc(num_tasks);

    /* Note, since we need to know the number of tasks before LAPI is          */
    /* initialized, we can't use LAPI_Qenv.  getenv("MP_PROCS") will           */
    /* do the trick.                                                           */


    /* populate addr_list                                                      */
    /* use the AIX(r) inet_addr call to convert an IP address                    */
    /* from a dotted quad to a long                                            */
    task_0_ip_as_long = inet_addr(task_0_ip_as_string);
    addr_list[0].ip_addr = task_0_ip_as_long;
    addr_list[0].port_no = task_0_port_as_unsigned;

    task_1_ip_as_long = inet_addr(task_1_ip_as_string);
    addr_list[1].ip_addr = task_1_ip_as_long;
    addr_list[1].port_no = task_1_port_as_unsigned;
    .
    .
    .
    task_num_tasks-1_ip_as_long = inet_addr(task_num_tasks-1_ip_as_string);
    addr_list[num_tasks-1].ip_addr = task_num_tasks-1_ip_as_long;
    addr_list[num_tasks-1].port_no = task_num_tasks-1_port_as_unsigned;


    /* then assign to extend pointer */
    extend_info.add_udp_addrs = addr_list;

    info.add_info = &extend_info;


    LAPI_Init(&hndl, &info);
    .
    .
    .

    /* user's responsibility only in the case of user list */
    free(addr_list);

}
```

See the LAPI sample programs for complete examples of initialization in standalone mode.

To initialize a LAPI handle for user space (US) communication in standalone mode:

```
export MP_MSG_API=lapi
export MP_EUILIB=us
export MP_PROCS=                  /* number of tasks in job   */
export MP_PARTITION=              /* unique job key           */
export MP_CHILD=                  /* unique task ID           */
```

```
export MP_LAPI_NETWORK=@1:164,sn0   /* LAPI network information */

run LAPI jobs as normal
```

See the README.LAPI.STANDALONE.US file in the standalone/us directory of the LAPI sample files for complete details.

Location

/usr/lib/liblapi_r.a

LAPI_Msg_string Subroutine

Purpose

Retrieves the message that is associated with a subroutine return code.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

LAPI_Msg_string(error_code, buf)
int error_code;
void *buf;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_MSG_STRING(error_code, buf, ierror)
INTEGER error_code
CHARACTER buf(LAPI_MAX_ERR_STRING)
INTEGER ierror
```

Description

Type of call: local queries

Use this subroutine to retrieve the message string that is associated with a LAPI return code. LAPI tries to find the messages of any return codes that come from the AIX operating system or its communication subsystem.

Parameters

INPUT

error_code
Specifies the return value of a previous LAPI call.

OUTPUT

buf
Specifies the buffer to store the message string.

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To get the message string associated with a LAPI return code:

```
{

    char msg_buf[LAPI_MAX_ERR_STRING]; /* constant defined in lapi.h */
    int rc, errc;

    rc = some_LAPI_call();

    errc = LAPI_Msg_string(rc, msg_buf);

    /* msg_buf now contains the message string for the return code   */

}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_CATALOG_FAIL
Indicates that the message catalog cannot be opened. An English-only string is copied into the user's message buffer (*buf*).

LAPI_ERR_CODE_UNKNOWN
Indicates that *error_code* is outside of the range known to LAPI.

LAPI_ERR_RET_PTR_NULL
Indicates that the value of the *buf* pointer is NULL (in C) or that the value of *buf* is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Msgpoll Subroutine

Purpose

Allows the calling thread to check communication progress.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Msgpoll(hndl, cnt, info)
lapi_handle_t     hndl;
uint              cnt;
lapi_msg_info_t  *info;


typedef struct {
    lapi_msg_state_t  status;      /* Message status returned from LAPI_Msgpoll */
    ulong             reserve[10]; /* Reserved                                  */
} lapi_msg_info_t;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_MSGPOLL(hndl, cnt, info, ierror)
INTEGER hndl
INTEGER cnt
TYPE (LAPI_MSG_STATE_T) :: info
INTEGER ierror
```

Description

`Type of call:` local progress monitor (blocking)

The `LAPI_Msgpoll` subroutine allows the calling thread to check communication progress. With this subroutine, LAPI provides a means of running the dispatcher several times until either progress is made or a specified maximum number of dispatcher loops have executed. Here, *progress* is defined as the completion of either a message send operation or a message receive operation.

`LAPI_Msgpoll` is intended to be used when interrupts are turned off. If the user has not explicitly turned interrupts off, LAPI temporarily disables interrupt mode while in this subroutine because the dispatcher is called, which will process any pending receive operations. If the LAPI dispatcher loops for the specified maximum number of times, the call returns. If progress is made before the maximum count, the call will return immediately. In either case, LAPI will report status through a data structure that is passed by reference.

The `lapi_msg_info_t` structure contains a flags field (*status*), which is of type `lapi_msg_state_t`. Flags in the *status* field are set as follows:

LAPI_DISP_CNTR
If the dispatcher has looped *cnt* times without making progress

LAPI_SEND_COMPLETE
If a message send operation has completed

LAPI_RECV_COMPLETE
If a message receive operation has completed

LAPI_BOTH_COMPLETE
If both a message send operation and a message receive operation have completed

LAPI_POLLING_NET
If another thread is already polling the network or shared memory completion

Parameters

INPUT

hndl
Specifies the LAPI handle.

cnt
Specifies the maximum number of times the dispatcher should loop with no progress before returning.

info
Specifies a status structure that contains the result of the `LAPI_Msgpoll()` call.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To loop through the dispatcher no more than 1000 times, then check what progress has been made:

```
{

    lapi_msg_info_t msg_info;
    int cnt = 1000;
    .
    .
    .
    LAPI_Msgpoll(hndl, cnt, &msg_info);

    if ( msg_info.status & LAPI_BOTH_COMPLETE ) {
        /* both a message receive and a message send have been completed */
    } else if ( msg_info.status & LAPI_RECV_COMPLETE ) {
        /* just a message receive has been completed                     */
    } else if ( msg_info.status & LAPI_SEND_COMPLETE ) {
        /* just a message send has been completed                        */
    } else {
        /* cnt loops and no progress                                     */
    }


}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_MSG_INFO_NULL
Indicates that the *info* pointer is NULL (in C) or that the value of *info* is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Nopoll_wait Subroutine

Purpose

Waits for a counter update without polling.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

void LAPI_Nopoll_wait(hndl, cntr_ptr, val, cur_cntr_val)
lapi_handle_t  hndl;
lapi_cntr_t   *cntr_ptr;
int            val;
int           *cur_cntr_val;
```

FORTRAN Syntax

```
include 'lapif.h'

int LAPI_NOPOLL_WAIT(hndl, cntr, val, cur_cntr_val, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
```

```
INTEGER val
INTEGER cur_cntr_val
INTEGER ierror
```

Description

`Type of call:` recovery (blocking)

This subroutine waits for a counter update without polling (that is, without explicitly invoking LAPI's internal communication dispatcher). This call may or may not check for message arrivals over the LAPI context *hndl*. The *cur_cntr_val* variable is set to the current counter value. Although it has higher latency than `LAPI_Waitcntr`, `LAPI_Nopoll_wait` frees up the processor for other uses.

Note: To use this subroutine, the *lib_vers* field in the `lapi_info_t` structure must be set to `L2_LIB` or `LAST_LIB`.

Parameters

INPUT

hndl
Specifies the LAPI handle.

val
Specifies the relative counter value (starting from 1) that the counter needs to reach before returning.

cur_cntr_val
Specifies the integer value of the current counter. The value of The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

INPUT/OUTPUT

cntr_ptr
Points to the `lapi_cntr_t` structure in C.

cntr
Is the `lapi_cntr_t` structure in FORTRAN.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL
Indicates that the *cntr_ptr* pointer is NULL (in C) or that the value of *cntr* is `LAPI_ADDR_NULL` (in FORTRAN).

LAPI_ERR_CNTR_VAL
Indicates that the *val* passed in is less than or equal to `0`.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_MULTIPLE_WAITERS
Indicates that more than one thread is waiting for the counter.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because `LAPI_Purge_totask()` was called.

Restrictions

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

Location

/usr/lib/liblapi_r.a

LAPI_Probe Subroutine

Purpose

Transfers control to the communication subsystem to check for arriving messages and to make progress in polling mode.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Probe(hndl)
lapi_handle_t hndl;
```

FORTRAN Syntax

```
include 'lapif.h'

int LAPI_PROBE(hndl, ierror)
INTEGER hndl
INTEGER ierror
```

Description

`Type of call:` local progress monitor (non-blocking)

This subroutine transfers control to the communication subsystem in order to make progress on messages associated with the context *hndl*. A `LAPI_Probe` operation lasts for one round of the communication dispatcher.

Note: There is no guarantee about receipt of messages on the return from this function.

Parameters

INPUT

hndl
Specifies the LAPI handle.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

LAPI_Purge_totask Subroutine

Purpose

Allows a task to cancel messages to a given destination.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Purge_totask(hndl, dest)
lapi_handle_t hndl;
uint          dest;
```

FORTRAN Syntax

```
include 'lapif.h'

int LAPI_PURGE_TOTASK(hndl, dest, ierror)
INTEGER hndl
INTEGER dest
INTEGER ierror
```

Description

`Type of call:` recovery

This subroutine cancels messages and resets the state corresponding to messages in flight or submitted to be sent to a particular target task. This is an entirely local operation. For correct behavior a similar invocation is expected on the destination (if it exists). This function cleans up all the state associated with pending messages to the indicated target task. It is assumed that before the indicated task starts communicating with this task again, it also purges this instance (or that it was terminated and initialized again). It will also wake up all threads that are in `LAPI_Nopoll_wait` depending on how the arguments are passed to the `LAPI_Nopoll_wait` function. The behavior of `LAPI_Purge_totask` is undefined if LAPI collective functions are used.

`Note:` This subroutine should not be used when the parallel application is running in a PE/LoadLeveler environment.

`LAPI_Purge_totask` is normally used after connectivity has been lost between two tasks. If connectivity is restored, the tasks can restored for LAPI communication by calling `LAPI_Resume_totask`.

Parameters

INPUT

hndl
Specifies the LAPI handle.

dest
Specifies the destination instance ID to which pending messages need to be cancelled.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is *not* recommended on a system that is running Parallel Environment (PE).

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_TGT
Indicates that *dest* is outside the range of tasks defined in the job.

Location

/usr/lib/liblapi_r.a

LAPI_Put Subroutine

Purpose

Transfers data from a local task to a remote task.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Put(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, cmpl_cntr)
lapi_handle_t hndl;
uint          tgt;
ulong         len;
void         *tgt_addr;
void         *org_addr;
lapi_cntr_t  *tgt_cntr;
lapi_cntr_t  *org_cntr;
lapi_cntr_t  *cmpl_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'

int LAPI_PUT(hndl, tgt, len, tgt_addr, org_addr, tgt_cntr, org_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_LONG_TYPE) :: len
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_addr
INTEGER org_addr
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

Description

`Type of call:` point-to-point communication (non-blocking)

Use this subroutine to transfer data from a local (origin) task to a remote (target) task. The origin counter will increment on the origin task upon origin buffer availability. The target counter will increment on the target and the completion counter will increment at the origin task upon message completion. Because there is no completion handler, message completion and target buffer availability are the same in this case.

This is a non-blocking call. The caller *cannot* assume that the data transfer has completed upon the return of the function. Instead, counters should be used to ensure correct buffer accesses as defined above.

Note that a zero-byte message does not transfer data, but it does have the same semantic with respect to counters as that of any other message.

Parameters

INPUT

hndl
Specifies the LAPI handle.

tgt
Specifies the task ID of the target task. The value of this parameter must be in the range `0` <= *tgt* < **NUM_TASKS**.

len
Specifies the number of bytes to be transferred. This parameter must be in the range `0` <= *len* <= the value of LAPI constant `LAPI_MAX_MSG_SZ`.

tgt_addr
Specifies the address on the target task where data is to be copied into. If *len* is `0`, The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

org_addr
Specifies the address on the origin task from which data is to be copied. If *len* is `0`, The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

INPUT/OUTPUT

tgt_cntr
Specifies the target counter address. The target counter is incremented upon message completion. If this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the target counter is not updated.

org_cntr
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the origin counter is not updated.

cmpl_cntr
Specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of *tgt_cntr*. The completion counter is incremented at the origin after *tgt_cntr* is incremented. If this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), the completion counter is not updated.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

```
{
    /* initialize the table buffer for the data addresses          */
    /* get remote data buffer addresses                            */
```

```
    LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
    .
    .
    .
    LAPI_Put(hndl, tgt, (ulong) data_len, (void *)(data_buffer_list[tgt]),
             (void *) data_buffer, tgt_cntr, org_cntr, compl_cntr);

    /* transfer data_len bytes from local address data_buffer.     */
    /* write the data starting at address data_buffer_list[tgt] on */
    /* task tgt. tgt_cntr, org_cntr, and compl_cntr can be NULL.   */

}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN
Indicates that the value of *len* is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_ADDR_NULL
Indicates that the *org_addr* parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but *len* is greater than 0.

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_ADDR_NULL
Indicates that the *tgt_addr* parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but *len* is greater than 0.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

Location

/usr/lib/liblapi_r.a

LAPI_Putv Subroutine

Purpose

Transfers vectors of data from a local task to a remote task.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, cmpl_cntr)

lapi_handle_t  hndl;
uint           tgt;
lapi_vec_t    *tgt_vec;
lapi_vec_t    *org_vec;
lapi_cntr_t   *tgt_cntr;
lapi_cntr_t   *org_cntr;
lapi_cntr_t   *cmpl_cntr;


typedef struct  {
```

```
    lapi_vectype_t  vec_type;   /* operation code        */
    uint            num_vecs;   /* number of vectors     */
    void            **info;     /* vector of information */
    ulong           *len;       /* vector of lengths     */
} lapi_vec_t;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_PUTV(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr , cmpl_cntr, ierror)
INTEGER hndl
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_vec
TYPE (LAPI_VEC_T) :: org_vec
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_cntr
TYPE (LAPI_CNTR_T) :: org_cntr
TYPE (LAPI_CNTR_T) :: cmpl_cntr
INTEGER ierror
```

The 32-bit version of the LAPI_VEC_T type is defined as:

```
TYPE LAPI_VEC_T
   SEQUENCE
   INTEGER(KIND = 4)  :: vec_type
   INTEGER(KIND = 4)  :: num_vecs
   INTEGER(KIND = 4)  :: info
   INTEGER(KIND = 4)  :: len
END TYPE LAPI_VEC_T
```

The 64-bit version of the LAPI_VEC_T type is defined as:

```
TYPE LAPI_VEC_T
   SEQUENCE
   INTEGER(KIND = 4)  :: vec_type
   INTEGER(KIND = 4)  :: num_vecs
   INTEGER(KIND = 8)  :: info
   INTEGER(KIND = 8)  :: len
END TYPE LAPI_VEC_T
```

Description

Type of call: point-to-point communication (non-blocking)

LAPI_Putv is the vector version of the LAPI_Put call. Use this subroutine to transfer vectors of data from the origin task to the target task. The origin vector descriptions and the target vector descriptions are located in the address space of the *origin* task. However, the values specified in the *info* array of the target vector must be addresses in the address space of the *target* task.

The calling program *cannot* assume that the origin buffer can be changed or that the contents of the target buffers on the target task are ready for use upon function return. After the origin counter (*org_cntr*) is incremented, the origin buffers can be modified by the origin task. After the target counter (*tgt_cntr*) is incremented, the target buffers can be modified by the target task. If you provide a completion counter (*cmpl_cntr*), it is incremented at the origin after the target counter (*tgt_cntr*) has been incremented at the target. If the values of any of the counters or counter addresses are NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the data transfer occurs, but the corresponding counter increments do not occur.

If a strided vector is being transferred, the size of each block must not be greater than the stride size in bytes.

The length of any vector pointed to by *org_vec* must be equal to the length of the corresponding vector pointed to by *tgt_vec*.

LAPI does not check for any overlapping regions among vectors either at the origin or the target. If the overlapping regions exist on the target side, the contents of the target buffer are undefined after the operation.

See LAPI_Amsendv for more information about using the various vector types. (LAPI_Putv does not support the LAPI_GEN_GENERIC type.)

Parameters

INPUT

hndl
Specifies the LAPI handle.

tgt
Specifies the task ID of the target task. The value of this parameter must be in the range 0 <= *tgt* < **NUM_TASKS**.

tgt_vec
Points to the target vector description.

org_vec
Points to the origin vector description.

INPUT/OUTPUT

tgt_cntr
Specifies the target counter address. The target counter is incremented upon message completion. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the target counter is not updated.

org_cntr
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). The origin counter is incremented at buffer availability. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the origin counter is not updated.

cmpl_cntr
Specifies the completion counter address (in C) or the completion counter (in FORTRAN) that is a reflection of *tgt_cntr*. The completion counter is incremented at the origin after *tgt_cntr* is incremented. If this parameter is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), the completion counter is not updated.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

C Examples

To put a LAPI_GEN_IOVECTOR:

```
{

      /* retrieve a remote data buffer address for data to transfer,  */
      /* such as through LAPI_Address_init                            */

      /* task that calls LAPI_Putv sets up both org_vec and tgt_vec   */
      org_vec->num_vecs = NUM_VECS;
      org_vec->vec_type = LAPI_GEN_IOVECTOR;
      org_vec->len      = (unsigned long *)
      malloc(NUM_VECS*sizeof(unsigned long));
      org_vec->info     = (void **) malloc(NUM_VECS*sizeof(void *));

      /* each org_vec->info[i] gets a base address on the origin task */
      /* each org_vec->len[i] gets the number of bytes to transfer    */
      /* from org_vec->info[i]                                        */

      tgt_vec->num_vecs = NUM_VECS;
      tgt_vec->vec_type = LAPI_GEN_IOVECTOR;
      tgt_vec->len      = (unsigned long *)
      malloc(NUM_VECS*sizeof(unsigned long));
      tgt_vec->info     = (void **) malloc(NUM_VECS*sizeof(void *));

      /* each tgt_vec->info[i] gets a base address on the target task           */
      /* each tgt_vec->len[i] gets the number of bytes to write to vec->info[i] */
      /* For LAPI_GEN_IOVECTOR, num_vecs, vec_type, and len must be the same    */
```

```
      LAPI_Putv(hndl, tgt, tgt_vec, org_vec, tgt_cntr, org_cntr, compl_cntr);
      /* tgt_cntr, org_cntr and compl_cntr can all be NULL */

      /* data will be transferred as follows:              */
      /* org_vec->len[0] bytes will be retrieved from      */
      /* org_vec->info[0] and written to tgt_vec->info[0]  */
      /* org_vec->len[1] bytes will be retrieved from      */
      /* org_vec->info[1] and written to tgt_vec->info[1]  */
      .
      .
      .
      /* org_vec->len[NUM_VECS-1] bytes will be retrieved */
      /* from org_vec->info[NUM_VECS-1] and written to    */
      /* tgt_vec->info[NUM_VECS-1]                        */

}
```

See the example in LAPI_Amsendv for information on other vector types.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_ORG_EXTENT
Indicates that the *org_vec*'s extent (stride * *num_vecs*) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE
Indicates that the *org_vec* stride is less than block.

LAPI_ERR_ORG_VEC_ADDR
Indicates that the *org_vec*->*info[i]* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (*org_vec*->*len[i]*) is not 0.

LAPI_ERR_ORG_VEC_LEN
Indicates that the sum of *org_vec*->*len* is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_VEC_NULL
Indicates that the *org_vec* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_ORG_VEC_TYPE
Indicates that the *org_vec*->*vec_type* is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL
Indicates that the strided vector address *org_vec*->*info*[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL
Indicates that the strided vector address *tgt_vec*->*info*[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_EXTENT
Indicates that *tgt_vec*'s extent (stride * *num_vecs*) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_TGT_STRIDE
Indicates that the *tgt_vec* stride is less than block.

LAPI_ERR_TGT_VEC_ADDR
Indicates that the *tgt_vec*->*info*[i] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (*tgt_vec*->*len*[i]) is not 0.

LAPI_ERR_TGT_VEC_LEN
Indicates that the sum of *tgt_vec->len* is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_VEC_NULL
Indicates that *tgt_vec* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_VEC_TYPE
Indicates that the *tgt_vec->vec_type* is not valid.

LAPI_ERR_VEC_LEN_DIFF
Indicates that *org_vec* and *tgt_vec* have different lengths (*len[ ]*).

LAPI_ERR_VEC_NUM_DIFF
Indicates that *org_vec* and *tgt_vec* have different *num_vecs*.

LAPI_ERR_VEC_TYPE_DIFF
Indicates that *org_vec* and *tgt_vec* have different vector types (*vec_type*).

Location

/usr/lib/liblapi_r.a

LAPI_Qenv Subroutine

Purpose

Used to query LAPI for runtime task information.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapif.h>

int LAPI_Qenv(hndl, query, ret_val)
lapi_handle_t  hndl;
lapi_query_t   query;
int           *ret_val; /* ret_val's type varies (see Additional query types) */
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_QENV(hndl, query, ret_val, ierror)
INTEGER hndl
INTEGER query
INTEGER ret_val /* ret_val's type varies (see Additional query types) */
INTEGER ierror
```

Description

Type of call: local queries

Use this subroutine to query runtime settings and statistics from LAPI. LAPI defines a set of query types as an enumeration in lapi.h for C and explicitly in the 32-bit and 64-bit versions of lapif.h for FORTRAN.

For example, you can query the size of the table that LAPI uses for the LAPI_Addr_set subroutine using a *query* value of LOC_ADDRTBL_SZ:

```
LAPI_Qenv(hndl, LOC_ADDRTBL_SZ, &ret_val);
```

ret_val will contain the upper bound on the table index. A subsequent call to `LAPI_Addr_set (`*hndl*`, `*addr*`, `*addr_hndl*`);` could then ensure that the value of *addr_hndl* is between `0` and *ret_val*.

When used to show the size of a parameter, a comparison of values, or a range of values, valid values for the *query* parameter of the `LAPI_Qenv` subroutine appear in **SMALL, BOLD** capital letters. For example:

NUM_TASKS

is a shorthand notation for:

`LAPI_Qenv(`*hndl*`, NUM_TASKS, `*ret_val*`)`

In C, `lapi_query_t` defines the valid types of LAPI queries:

```
typedef enum {
        TASK_ID=0,      /* Query the task ID of the current task in the job        */
        NUM_TASKS,      /* Query the number of tasks in the job                    */
        MAX_UHDR_SZ,    /* Query the maximum user header size for active messaging */
        MAX_DATA_SZ,    /* Query the maximum data length that can be sent          */
        ERROR_CHK,      /* Query and set parameter checking on (1) or off (0)      */
        TIMEOUT,        /* Query and set the current communication timeout setting
*/
                        /* in seconds                                              */
        MIN_TIMEOUT, /* Query the minimum communication timeout setting in seconds
*/
        MAX_TIMEOUT, /* Query the maximum communication timeout setting in seconds
*/
        INTERRUPT_SET,  /* Query and set interrupt mode on (1) or off (0)          */
        MAX_PORTS,    /* Query the maximum number of available communication ports */
        MAX_PKT_SZ,     /* This is the payload size of 1 packet                    */
        NUM_REX_BUFS,   /* Number of retransmission buffers                        */
        REX_BUF_SZ,     /* Size of each retransmission buffer in bytes             */
        LOC_ADDRTBL_SZ, /* Size of address store table used by LAPI_Addr_set       */
        EPOCH_NUM,      /* No longer used by LAPI (supports legacy code)           */
        USE_THRESH,     /* No longer used by LAPI (supports legacy code)           */
        RCV_FIFO_SIZE,  /* No longer used by LAPI (supports legacy code)           */
        MAX_ATOM_SIZE,/* Query the maximum atom size for a DGSP accumulate transfer*/
        BUF_CP_SIZE, /* Query the size of the message buffer to save (default 128b)*/
        MAX_PKTS_OUT,   /* Query the maximum number of messages outstanding /
*/
                        /* destination                                             */
        ACK_THRESHOLD,  /* Query and set the threshold of acknowledgments going
*/
                        /* back to the source                                      */
        QUERY_SHM_ENABLED,    /* Query to see if shared memory is enabled          */
        QUERY_SHM_NUM_TASKS,  /* Query to get the number of tasks that use shared
*/
                           /* memory                                               */
        QUERY_SHM_TASKS,  /* Query to get the list of task IDs that make up shared */
                          /* memory; pass in an array of size QUERY_SHM_NUM_TASKS  */
        QUERY_STATISTICS,    /* Query to get packet statistics from LAPI, as       */
                             /* defined by the lapi_statistics_t structure. For    */
                             /* this query, pass in 'lapi_statistics_t *' rather   */
                             /* than 'int *ret_val'; otherwise, the data will      */
                             /* overflow the buffer.                               */
        PRINT_STATISTICS,    /* Query debug print function to print out statistics */
        QUERY_SHM_STATISTICS,/* Similar query as QUERY_STATISTICS for shared       */
                             /* memory path.                                       */
        QUERY_LOCAL_SEND_STATISTICS ,/* Similar query as QUERY_STATISTICS          */
                                     /* for local copy path.                       */
        BULK_XFER, /* Query to see if bulk transfer is enabled (1) or disabled (0) */
        BULK_MIN_MSG_SIZE, /* Query the current bulk transfer minimum message size
*/
        LAST_QUERY
} lapi_query_t;


typedef struct {
```

```
    lapi_long_t Tot_dup_pkt_cnt;     /* Total duplicate packet count  */
    lapi_long_t Tot_retrans_pkt_cnt; /* Total retransmit packet count */
    lapi_long_t Tot_gho_pkt_cnt;     /* Total Ghost packet count      */
    lapi_long_t Tot_pkt_sent_cnt;    /* Total packet sent count       */
    lapi_long_t Tot_pkt_recv_cnt;    /* Total packet receive count    */
    lapi_long_t Tot_data_sent;       /* Total data sent               */
    lapi_long_t Tot_data_recv;       /* Total data receive            */
} lapi_statistics_t;
```

In FORTRAN, the valid types of LAPI queries are defined in `lapif.h` as follows:

```
integer TASK_ID,NUM_TASKS,MAX_UHDR_SZ,MAX_DATA_SZ,ERROR_CHK
integer TIMEOUT,MIN_TIMEOUT,MAX_TIMEOUT
integer INTERRUPT_SET,MAX_PORTS,MAX_PKT_SZ,NUM_REX_BUFS
integer REX_BUF_SZ,LOC_ADDRTBL_SZ,EPOCH_NUM,USE_THRESH
integer RCV_FIFO_SIZE,MAX_ATOM_SIZE,BUF_CP_SIZE
integer MAX_PKTS_OUT,ACK_THRESHOLD,QUERY_SHM_ENABLED
integer QUERY_SHM_NUM_TASKS,QUERY_SHM_TASKS
integer QUERY_STATISTICS,PRINT_STATISTICS
integer QUERY_SHM_STATISTICS,QUERY_LOCAL_SEND_STATISTICS
integer BULK_XFER,BULK_MIN_MSG_SIZE,
integer LAST_QUERY
parameter (TASK_ID=0,NUM_TASKS=1,MAX_UHDR_SZ=2,MAX_DATA_SZ=3)
parameter (ERROR_CHK=4,TIMEOUT=5,MIN_TIMEOUT=6)
parameter (MAX_TIMEOUT=7,INTERRUPT_SET=8,MAX_PORTS=9)
parameter (MAX_PKT_SZ=10,NUM_REX_BUFS=11,REX_BUF_SZ=12)
parameter (LOC_ADDRTBL_SZ=13,EPOCH_NUM=14,USE_THRESH=15)
parameter (RCV_FIFO_SIZE=16,MAX_ATOM_SIZE=17,BUF_CP_SIZE=18)
parameter (MAX_PKTS_OUT=19,ACK_THRESHOLD=20)
parameter (QUERY_SHM_ENABLED=21,QUERY_SHM_NUM_TASKS=22)
parameter (QUERY_SHM_TASKS=23,QUERY_STATISTICS=24)
parameter (PRINT_STATISTICS=25)
parameter (QUERY_SHM_STATISTICS=26,QUERY_LOCAL_SEND_STATISTICS=27)
parameter (BULK_XFER=28,BULK_MIN_MSG_SIZE=29)
parameter (LAST_QUERY=30)
```

Additional query types

LAPI provides additional query types for which the behavior of LAPI_Qenv is slightly different:

PRINT_STATISTICS
When passed this query type, LAPI sends data transfer statistics to standard output. In this case, *ret_val* is unaffected. However, LAPI's error checking requires that the value of *ret_val* is not NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) for all LAPI_Qenv types (including PRINT_STATISTICS).

QUERY_LOCAL_SEND_STATISTICS
When passed this query type, LAPI_Qenv interprets *ret_val* as a pointer to type `lapi_statistics_t`. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through intra-task local copy. The packet count will be 0.

QUERY_SHM_STATISTICS
When passed this query type, LAPI_Qenv interprets *ret_val* as a pointer to type `lapi_statistics_t`. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred through shared memory.

QUERY_SHM_TASKS
When passed this query type, LAPI_Qenv returns a list of task IDs with which this task can communicate using shared memory. *ret_val* must be an `int *` with enough space to hold NUM_TASKS integers. For each task *i*, if it is possible to use shared memory, *ret_val*[*i*] will contain the shared memory task ID. If it is not possible to use shared memory, *ret_val*[*i*] will contain -1.

QUERY_STATISTICS
When passed this query type, LAPI_Qenv interprets *ret_val* as a pointer to type `lapi_statistics_t`. Upon function return, the fields of the structure contain LAPI's data transfer statistics for data transferred using the user space (US) protocol or UDP/IP.

Parameters

INPUT

hndl
Specifies the LAPI handle.

query
Specifies the type of query you want to request. In C, the values for *query* are defined by the `lapi_query_t` enumeration in `lapi.h`. In FORTRAN, these values are defined explicitly in the 32-bit version and the 64-bit version of `lapif.h`.

OUTPUT

ret_val
Specifies the reference parameter for LAPI to store as the result of the query. The value of this parameter cannot be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_QUERY_TYPE
Indicates that the query passed in is not valid.

LAPI_ERR_RET_PTR_NULL
Indicates that the value of the *ret_val* pointer is NULL (in C) or that the value of *ret_val* is `LAPI_ADDR_NULL` (in FORTRAN).

C Examples

To query runtime values from LAPI:

```
{
    int               task_id;
    lapi_statistics_t stats;
    .
    .
    .
    LAPI_Qenv(hndl, TASK_ID, &task_id);
    /* task_id now contains the task ID */
    .
    .
    .
    LAPI_Qenv(hndl, QUERY_STATISTICS, (int *)&stats);
    /* the fields of the stats structure are now
       filled in with runtime values */
    .
    .
    .
}
```

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: `LAPI_Amsend`, `LAPI_Get`, `LAPI_Put`, `LAPI_Senv`, `LAPI_Xfer`

LAPI_Resume_totask Subroutine

Purpose

Re-enables the sending of messages to the task.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Resume_totask(hndl, dest)
lapi_handle_t hndl;
uint    dest;
```

FORTRAN Syntax

```
include 'lapif.h'

int LAPI_RESUME_TOTASK(hndl, dest, ierror)
INTEGER hndl
INTEGER dest
INTEGER ierror
```

Description

`Type of call:` recovery

This subroutine is used in conjunction with `LAPI_Purge_totask`. It enables LAPI communication to be reestablished for a task that had previously been purged. The purged task must either restart LAPI or execute a `LAPI_Purge_totask`/`LAPI_Resume_totask` sequence for this task.

Parameters

INPUT

hndl
Specifies the LAPI handle.

dest
Specifies the destination instance ID with which to resume communication.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is *not* recommmended on a system that is running Parallel Environment (PE).

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

Location

/usr/lib/liblapi_r.a

LAPI_Rmw Subroutine

Purpose

Provides data synchronization primitives.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Rmw(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)

lapi_handle_t hndl;
RMW_ops_t op;
uint tgt;
int *tgt_var;
int *in_val;
int *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_RMW(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr, ierror)
INTEGER hndl
INTEGER op
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_var
INTEGER in_val
INTEGER prev_tgt_val
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

Description

`Type of call:` point-to-point communication (non-blocking)

Use this subroutine to synchronize two independent pieces of data, such as two tasks sharing a common data structure. The operation is performed at the target task (*tgt*) and is atomic. The operation takes an input value (*in_val*) from the origin and performs one of four operations (*op*) on a variable (*tgt_var*) at the target (*tgt*), and then replaces the target variable (*tgt_var*) with the results of the operation (*op*). The original value (*prev_tgt_val*) of the target variable (*tgt_var*) is returned to the origin.

The operations (*op*) are performed over the context referred to by *hndl*. The outcome of the execution of these calls is as if the following code was executed atomically:

```
*prev_tgt_val = *tgt_var;
*tgt_var      = f(*tgt_var, *in_val);
```

where:

f(a,b) = a + b for **FETCH_AND_ADD**

f(a,b) = a | b for **FETCH_AND_OR** (bitwise or)

f(a,b) = b for SWAP

For **COMPARE_AND_SWAP**, *in_val* is treated as a pointer to an array of two integers, and the *op* is the following atomic operation:

```
 if(*tgt_var == in_val[0]) {
   *prev_tgt_val = TRUE;
   *tgt_var      = in_val[1];
} else {
   *prev_tgt_val = FALSE;
}
```

All LAPI_Rmw calls are non-blocking. To test for completion, use the LAPI_Getcntr and LAPI_Waitcntr subroutines. LAPI_Rmw does not include a target counter (*tgt_cntr*), so LAPI_Rmw calls do not provide any indication of completion on the target task (*tgt*).

Parameters

INPUT

hndl
Specifies the LAPI handle.

op
Specifies the operation to be performed. The valid operations are:

- COMPARE_AND_SWAP
- FETCH_AND_ADD
- FETCH_AND_OR
- SWAP

tgt
Specifies the task ID of the target task where the read-modify-write (Rmw) variable resides. The value of this parameter must be in the range 0 <= *tgt* < **NUM_TASKS**.

tgt_var
Specifies the target read-modify-write (Rmw) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

in_val
Specifies the value that is passed in to the operation (*op*). This value cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

prev_tgt_val
Specifies the location at the origin in which the previous *tgt_var* on the target task is stored before the operation (*op*) is executed. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

org_cntr
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If *prev_tgt_val* is set, the origin counter (*org_cntr*) is incremented when *prev_tgt_val* is returned to the origin side. If *prev_tgt_val* is not set, the origin counter (*org_cntr*) is updated after the operation (*op*) is completed at the target side.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

1. To synchronize a data value between two tasks (with FETCH_AND_ADD):

```
{

       int local_var;
       int *addr_list;

       /* both tasks initialize local_var to a value       */

       /* local_var addresses are exchanged and stored     */
       /* in addr_list (using LAPI_Address_init).          */
       /* addr_list[tgt] now contains the address of       */
       /* local_var on tgt                                 */
       .
       .
       .
       /* add value to local_var on some task               */

       /* use LAPI to add value to local_var on remote task */
       LAPI_Rmw(hndl, FETCH_AND_ADD, tgt, addr_list[tgt],
                value, prev_tgt_val, &org_cntr);

       /* local_var on the remote task has been increased   */
       /* by value.  prev_tgt_val now contains the value    */
       /* of local_var on remote task before the addition   */

}
```

2. To synchronize a data value between two tasks (with SWAP):

```
{

      int local_var;
      int *addr_list;

      /* local_var addresses are exchanged and stored  */
      /* in addr_list (using LAPI_Address_init).       */
      /* addr_list[tgt] now contains the address of    */
      /* local_var on tgt.                             */
      .
      .
      .
      /* local_var is assigned some value              */

      /* assign local_var to local_var on remote task  */
      LAPI_Rmw(hndl, SWAP, tgt, addr_list[tgt],
               local_var, prev_tgt_val, &org_cntr);

      /* local_var on the remote task is now equal to  */
      /* local_var on the local task. prev_tgt_val now */
      /* contains the value of local_var on the remote */
      /* task before the swap.                         */

}
```

3. To conditionally swap a data value (with COMPARE_AND_SWAP):

```
{

      int local_var;
      int *addr_list;
      int in_val[2];

      /* local_var addresses are exchanged and stored        */
      /* in addr_list (using LAPI_Address_init).             */
      /* addr_list[tgt] now contains the address of          */
      /* local_var on tgt.                                   */
      .
      .
      .
      /* if local_var on remote_task is equal to comparator, */
      /* assign value to local_var on remote task            */

      in_val[0] = comparator;
      in_val[1] = value;
```

```
      LAPI_Rmw(hndl, COMPARE_AND_SWAP, tgt, addr_list[tgt],
               in_val, prev_tgt_val, &org_cntr);

      /* local_var on the remote task is now in_val[1] if it */
      /* had previously been equal to in_val[0]. If the swap */
      /* was performed, prev_tgt_val now contains TRUE;      */
      /* otherwise, it contains FALSE.                       */

}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_IN_VAL_NULL
Indicates that the *in_val* pointer is NULL (in C) or that the value of *in_val* is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_RMW_OP
Indicates that *op* is not valid.

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_TGT_VAR_NULL
Indicates that the tgt_var address is NULL (in C) or that the value of tgt_var is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Rmw64 Subroutine

Purpose

Provides data synchronization primitives for 64-bit applications.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Rmw64(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr)

lapi_handle_t hndl;
Rmw_ops_t op;
uint tgt;
long long *tgt_var;
long long *in_val;
long long *prev_tgt_val;
lapi_cntr_t *org_cntr;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_RMW64(hndl, op, tgt, tgt_var, in_val, prev_tgt_val, org_cntr,  ierror)

INTEGER hndl
INTEGER op
INTEGER tgt
INTEGER (KIND=LAPI_ADDR_TYPE) :: tgt_var
INTEGER (KIND=LAPI_LONG_LONG_TYPE) :: in_val, prev_tgt_val
TYPE (LAPI_CNTR_T) :: org_cntr
INTEGER ierror
```

Description

`Type of call:` point-to-point communication (non-blocking)

This subroutine is the 64-bit version of `LAPI_Rmw`. It is used to synchronize two independent pieces of 64-bit data, such as two tasks sharing a common data structure. The operation is performed at the target task (*tgt*) and is atomic. The operation takes an input value (*in_val*) from the origin and performs one of four operations (*op*) on a variable (*tgt_var*) at the target (*tgt*), and then replaces the target variable (*tgt_var*) with the results of the operation (*op*). The original value (*prev_tgt_val*) of the target variable (*tgt_var*) is returned to the origin.

The operations (*op*) are performed over the context referred to by *hndl*. The outcome of the execution of these calls is as if the following code was executed atomically:

```
*prev_tgt_val = *tgt_var;
*tgt_var      = f(*tgt_var, *in_val);
```

where:

f(a,b) = a + b for **FETCH_AND_ADD**

f(a,b) = a | b for **FETCH_AND_OR** (bitwise or)

f(a,b) = b for `SWAP`

For `COMPARE_AND_SWAP`, *in_val* is treated as a pointer to an array of two integers, and the *op* is the following atomic operation:

```
if(*tgt_var == in_val[0])  {
   *prev_tgt_val = TRUE;
   *tgt_var      = in_val[1];
} else {
   *prev_tgt_val = FALSE;
}
```

This subroutine can also be used on a 32-bit processor.

All `LAPI_Rmw64` calls are non-blocking. To test for completion, use the `LAPI_Getcntr` and `LAPI_Waitcntr` subroutines. `LAPI_Rmw64` does not include a target counter (*tgt_cntr*), so `LAPI_Rmw64` calls do not provide any indication of completion on the target task (*tgt*).

Parameters

INPUT

hndl
Specifies the LAPI handle.

op
Specifies the operation to be performed. The valid operations are:

- `COMPARE_AND_SWAP`

- FETCH_AND_ADD
- FETCH_AND_OR
- SWAP

tgt
Specifies the task ID of the target task where the read-modify-write (Rmw64) variable resides. The value of this parameter must be in the range 0 <= *tgt* < **NUM_TASKS**.

tgt_var
Specifies the target read-modify-write (Rmw64) variable (in FORTRAN) or its address (in C). The value of this parameter cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

in_val
Specifies the value that is passed in to the operation (*op*). This value cannot be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

INPUT/OUTPUT

prev_tgt_val
Specifies the location at the origin in which the previous *tgt_var* on the target task is stored before the operation (*op*) is executed. The value of this parameter can be NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

org_cntr
Specifies the origin counter address (in C) or the origin counter (in FORTRAN). If *prev_tgt_val* is set, the origin counter (*org_cntr*) is incremented when *prev_tgt_val* is returned to the origin side. If *prev_tgt_val* is not set, the origin counter (*org_cntr*) is updated after the operation (*op*) is completed at the target side.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

1. To synchronize a data value between two tasks (with FETCH_AND_ADD):

```
{

      long long local_var;
      long long *addr_list;

      /* both tasks initialize local_var to a value        */

      /* local_var addresses are exchanged and stored      */
      /* in addr_list (using LAPI_Address_init64)          */
      /* addr_list[tgt] now contains address of            */
      /* local_var on tgt                                  */
      .
      .
      .
      /* add value to local_var on some task               */

      /* use LAPI to add value to local_var on remote task */
      LAPI_Rmw64(hndl, FETCH_AND_ADD, tgt, addr_list[tgt],
                 value, prev_tgt_val, &org_cntr);

      /* local_var on remote task has been increased       */
      /* by value.  prev_tgt_val now contains value of     */
      /* local_var on remote task before the addition      */

}
```

2. To synchronize a data value between two tasks (with SWAP):

```
{

      long long local_var;
      long long *addr_list;

      /* local_var addresses are exchanged and stored          */
      /* in addr_list (using LAPI_Address_init64).             */
      /* addr_list[tgt] now contains the address of            */
      /* local_var on tgt.                                     */
      .
      .
      .
      /* local_var is assigned some value                      */

      /* assign local_var to local_var on the remote task      */
      LAPI_Rmw64(hndl, SWAP, tgt, addr_list[tgt],
                 local_var, prev_tgt_val, &org_cntr);

      /* local_var on the remote task is now equal to local_var */
      /* on the local task. prev_tgt_val now contains the value */
      /* of local_var on the remote task before the swap.      */

}
```

3. To conditionally swap a data value (with COMPARE_AND_SWAP):

```
{

      long long local_var;
      long long *addr_list;
      long long in_val[2];

      /* local_var addresses are exchanged and stored        */
      /* in addr_list (using LAPI_Address_init64).           */
      /* addr_list[tgt] now contains the address of          */
      /* local_var on tgt.                                   */
      .
      .
      .
      /* if local_var on remote_task is equal to comparator, */
      /* assign value to local_var on the remote task        */

      in_val[0] = comparator;
      in_val[1] = value;

      LAPI_Rmw64(hndl, COMPARE_AND_SWAP, tgt, addr_list[tgt],
                 in_val, prev_tgt_val, &org_cntr);

      /* local_var on remote task is now in_val[1] if it     */
      /* had previously been equal to in_val[0]. If the      */
      /* swap was performed, prev_tgt_val now contains       */
      /* TRUE; otherwise, it contains FALSE.                 */

}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_IN_VAL_NULL
Indicates that the *in_val* pointer is NULL (in C) or that the value of *in_val* is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_RMW_OP
Indicates that *op* is not valid.

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_TGT_VAR_NULL
Indicates that the tgt_var address is NULL (in C) or that the value of tgt_var is LAPI_ADDR_NULL (in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Senv Subroutine

Purpose

Used to set a runtime variable.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapif.h>

int LAPI_Senv(hndl, query, set_val)
lapi_handle_t hndl;
lapi_query_t query;
int set_val;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_SENV(hndl, query, set_val, ierror)
INTEGER hndl
INTEGER query
INTEGER set_val
INTEGER ierror
```

Description

Type of call: local queries

Use this subroutine to set runtime attributes for a specific LAPI instance. In C, the lapi_query_t enumeration defines the attributes that can be set at runtime. These attributes are defined explicitly in FORTRAN. See LAPI_Qenv for more information.

You can use LAPI_Senv to set these runtime attributes: ACK_THRESHOLD, ERROR_CHK, INTERRUPT_SET, and TIMEOUT.

Parameters

INPUT

hndl
Specifies the LAPI handle.

query
Specifies the type of query that you want to set. In C, the values for *query* are defined by the lapi_query_t enumeration in lapi.h. In FORTRAN, these values are defined explicitly in the 32-bit version and the 64-bit version of lapif.h.

set_val
Specifies the integer value of the query that you want to set.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

The following values can be set using LAPI_Senv:

```
ACK_THRESHOLD:
int value;
LAPI_Senv(hndl, ACK_THRESHOLD, value);
/* LAPI sends packet acknowledgements (acks) in groups, waiting until  */
/* ACK_THRESHOLD packets have arrived before returning a group of acks */
/* The valid range for ACK_THRESHOLD is (1 <= value <= 30)             */
/* The default is 30.                                                  */

ERROR_CHK:
boolean toggle;
LAPI_Senv(hndl, ERROR_CHK, toggle);
/* Indicates whether LAPI should perform error checking.  If set, LAPI */
/* calls will perform bounds-checking on parameters.  Error checking   */
/* is disabled by default.                                             */

INTERRUPT_SET:
boolean toggle;
LAPI_Senv(hndl, INTERRUPT_SET, toggle);
/* Determines whether LAPI will respond to interrupts.  If interrupts  */
/* are disabled, LAPI will poll for message completion.                */
/* toggle==True will enable interrupts, False will disable.            */
/* Interrupts are enabled by default.                                  */

TIMEOUT:
int value;
LAPI_Senv(hndl, TIMEOUT, value);
/* LAPI will time out on a communication if no response is received    */
/* within timeout seconds.  Valid range is (10 <= timeout <= 86400).   */
/* 86400 seconds = 24 hours.  Default value is 900 (15 minutes).       */
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_QUERY_TYPE
Indicates the query passed in is not valid.

LAPI_ERR_SET_VAL
Indicates the set_val pointer is not in valid range.

Location

/usr/lib/liblapi_r.a

LAPI_Setcntr Subroutine

Purpose

Used to set a counter to a specified value.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Setcntr(hndl, cntr, val)
lapi_handle_t hndl;
lapi_cntr_t  *cntr;
int           val;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_SETCNTR(hndl, cntr, val, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER ierror
```

Description

`Type of call:` Local counter manipulation

This subroutine sets *cntr* to the value specified by *val*. Because the `LAPI_Getcntr`/`LAPI_Setcntr` sequence cannot be made atomic, you should only use `LAPI_Setcntr` when you know there will not be any competing operations.

Parameters

INPUT

hndl
Specifies the LAPI handle.

val
Specifies the value to which the counter needs to be set.

INPUT/OUTPUT

cntr
Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

To initialize a counter for use in a communication API call:

```
{
    lapi_cntr_t   my_tgt_cntr, *tgt_cntr_array;
    int           initial_value, expected_value, current_value;
    lapi_handle_t hndl;
    .
    .
    .
    /*
     * Note: the code below is executed on all tasks
     */

    /* initialize, allocate and create structures */
    initial_value  = 0;
    expected_value = 1;

    /* set the cntr to zero */
    LAPI_Setcntr(hndl, &my_tgt_cntr, initial_value);
    /* set other counters   */
    .
    .
    .
    /* exchange counter addresses, LAPI_Address_init synchronizes */
    LAPI_Address_init(hndl, &my_tgt_cntr, tgt_cntr_array);
    /* more address exchanges */
    .
    .
    .
    /* Communication calls using my_tgt_cntr */
    LAPI_Put(....., tgt_cntr_array[tgt], ....);
    .
    .
    .
    /* Wait for counter to reach value */
    for (;;) {
        LAPI_Getcntr(hndl, &my_tgt_cntr, &current_value);
        if (current_value >= expected_value) {
            break; /* out of infinite loop */
        } else {
            LAPI_Probe(hndl);
        }
    }
    .
    .
    .
    /* Quiesce/synchronize to ensure communication using our counter is done */
    LAPI_Gfence(hndl);
    /* Reset the counter */
    LAPI_Setcntr(hndl, &my_tgt_cntr, initial_value);
    /*
     * Synchronize again so that no other communication using the counter can
     * begin from any other task until we're all finished resetting the counter.
     */
    LAPI_Gfence(hndl);

    /* More communication calls */
    .
    .
    .
}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL
Indicates that the *cntr* value passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

LAPI_Setcntr_wstatus Subroutine

Purpose

Used to set a counter to a specified value and to set the associated destination list array and destination status array to the counter.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Setcntr_wstatus(hndl, cntr, num_dest, dest_list, dest_status)

lapi_handle_t  hndl;
lapi_cntr_t   *cntr;
int            num_dest;
uint          *dest_list;
int           *dest_status;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_SETCNTR_WSTATUS(hndl, cntr, num_dest, dest_list, dest_status, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER num_dest
INTEGER dest_list(*)
INTEGER dest_status
INTEGER ierror
```

Description

`Type of call:` recovery

This subroutine sets *cntr* to `0`. Use `LAPI_Setcntr_wstatus` to set the associated destination list array (*dest_list*) and destination status array (*dest_status*) to the counter. Use a corresponding `LAPI_Nopoll_wait` call to access these arrays. These arrays record the status of a task from where the thread calling `LAPI_Nopoll_wait()` is waiting for a response.

The return values for *dest_status* are:

LAPI_MSG_INITIAL
The task is purged or is not received.

LAPI_MSG_RECVD
The task is received.

LAPI_MSG_PURGED
The task is purged, but not received.

LAPI_MSG_PURGED_RCVD
The task is received and then purged.

LAPI_MSG_INVALID
Not valid; the task is already purged.

Note: To use this subroutine, the *lib_vers* field in the `lapi_info_t` structure must be set to `L2_LIB` or `LAST_LIB`.

Parameters

INPUT

hndl
Specifies the LAPI handle.

num_dest
Specifies the number of tasks in the destination list.

dest_list
Specifies an array of destinations waiting for this counter update. If the value of this parameter is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN), no status is returned to the user.

INPUT/OUTPUT

cntr
Specifies the address of the counter to be set (in C) or the counter structure (in FORTRAN). The value of this parameter cannot be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

OUTPUT

dest_status
Specifies an array of status that corresponds to *dest_list*. The value of this parameter can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

Use of this subroutine is *not* recommmended on a system that is running Parallel Environment (PE).

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL
Indicates that the *cntr* value passed in is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_RET_PTR_NULL
Indicates that the value of *dest_status* is NULL in C (or `LAPI_ADDR_NULL` in FORTRAN), but the value of *dest_list* is not NULL in C (or `LAPI_ADDR_NULL` in FORTRAN).

Location

/usr/lib/liblapi_r.a

LAPI_Term Subroutine

Purpose

Terminates and cleans up a LAPI context.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Term(hndl)
lapi_handle_t hndl;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_TERM(hndl, ierror)
INTEGER hndl
INTEGER ierror
```

Description

`Type of call:` local termination

Use this subroutine to terminate the LAPI context that is specified by *hndl*. Any LAPI notification threads that are associated with this context are terminated. An error occurs when any LAPI calls are made using *hndl* after `LAPI_Term` is called.

A DGSP that is registered under that LAPI handle remains valid even after `LAPI_Term` is called on *hndl*.

Parameters

INPUT

hndl
Specifies the LAPI handle.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

To terminate a LAPI context (represented by *hndl*):

```
LAPI_Term(hndl);
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

LAPI_Util Subroutine

Purpose

Serves as a wrapper function for such data gather/scatter operations as registration and reservation, for updating UDP port information, and for obtaining pointers to locking and signaling functions that are associated with a shared LAPI lock.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Util(hndl, util_cmd)
lapi_handle_t hndl;
lapi_util_t *util_cmd;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_UTIL(hndl, util_cmd, ierror)
INTEGER hndl
TYPE (LAPI_UTIL_T) :: util_cmd
INTEGER ierror
```

Description

`Type of call:` Data gather/scatter program (DGSP), UDP port information, and lock sharing utilities

This subroutine is used for several different operations, which are indicated by the command type value in the beginning of the command structure. The `lapi_util_t` structure is defined as:

```
typedef union {
    lapi_util_type_t    Util_type;
    lapi_reg_dgsp_t     RegDgsp;
    lapi_dref_dgsp_t    DrefDgsp;
    lapi_resv_dgsp_t    ResvDgsp;
    lapi_reg_ddm_t      DdmFunc;
    lapi_add_udp_port_t Udp;
    lapi_pack_dgsp_t    PackDgsp;
    lapi_unpack_dgsp_t  UnpackDgsp;
    lapi_thread_func_t  ThreadFunc;
} lapi_util_t;
```

The enumerated type `lapi_util_type_t` has these values:

Table 2. lapi_util_type_t types

Value of *Util_type*	**Union member as interpreted by LAPI_Util**
LAPI_REGISTER_DGSP	lapi_reg_dgsp_t
LAPI_UNRESERVE_DGSP	lapi_dref_dgsp_t
LAPI_RESERVE_DGSP	lapi_resv_dgsp_t
LAPI_REG_DDM_FUNC	lapi_reg_ddm_t
LAPI_ADD_UDP_DEST_PORT	lapi_add_udp_port_t
LAPI_DGSP_PACK	lapi_pack_dgsp_t

Table 2. lapi_util_type_t types (continued)	
Value of *Util_type*	**Union member as interpreted by LAPI_Util**
LAPI_DGSP_UNPACK	lapi_unpack_dgsp_t
LAPI_GET_THREAD_FUNC	lapi_thread_func_t

hndl is not checked for command type LAPI_REGISTER_DGSP, LAPI_RESERVE_DGSP, or LAPI_UNRESERVE_DGSP.

LAPI_REGISTER_DGSP

You can use this operation to register a LAPI DGSP that you have created. To register a LAPI DGSP, lapi_dgsp_descr_t *idgsp* must be passed in. LAPI returns a handle (lapi_dg_handle_t *dgsp_handle*) to use for all future LAPI calls. The *dgsp_handle* that is returned by a register operation is identified as a lapi_dg_handle_t type, which is the appropriate type for LAPI_Xfer and LAPI_Util calls that take a DGSP. This returned *dgsp_handle* is also defined to be castable to a pointer to a lapi_dgsp_descr_t for those situations where the LAPI user requires read-only access to information that is contained in the cached DGSP. The register operation delivers a DGSP to LAPI for use in future message send, receive, pack, and unpack operations. LAPI creates its own copy of the DGSP and protects it by reference count. All internal LAPI operations that depend on a DGSP cached in LAPI ensure the preservation of the DGSP by incrementing the reference count when they begin a dependency on the DGSP and decrementing the count when that dependency ends. A DGSP, once registered, can be used from any LAPI instance. LAPI_Term does not discard any DGSPs.

You can register a DGSP, start one or more LAPI operations using the DGSP, and then unreserve it with no concern about when the LAPI operations that depend on the DGSP will be done using it. See LAPI_RESERVE_DGSP and LAPI_UNRESERVE_DGSP for more information.

In general, the DGSP you create and pass in to the LAPI_REGISTER_DGSP call using the *dgsp* parameter is discarded after LAPI makes and caches its own copy. Because DGSP creation is complex, user errors may occur, but extensive error checking at data transfer time would hurt performance. When developing code that creates DGSPs, you can invoke extra validation at the point of registration by setting the LAPI_VERIFY_DGSP environment variable. LAPI_Util will return any detected errors. Any errors that exist and are not detected at registration time will cause problems during data transfer. Any errors detected during data transfer will be reported by an asynchronous error handler. A segmentation fault is one common symptom of a faulty DGSP. If multiple DGSPs are in use, the asynchronous error handler will not be able to identify which DGSP caused the error. For more information about asynchronous error handling, see LAPI_Init.

LAPI_REGISTER_DGSP uses the lapi_reg_dgsp_t command structure.

Table 3. The lapi_reg_dgsp_t fields		
lapi_reg_dgsp_t field	**lapi_reg_dgsp_t field type**	**lapi_reg_dgsp_t usage**
Util_type	lapi_util_type_t	LAPI_REGISTER_DGSP
idgsp	lapi_dgsp_descr_t	IN - pointer to DGSP program
dgsp_handle	lapi_dg_handle_t	OUT - handle for a registered DGSP program
in_usr_func	lapi_usr_fcall_t	For debugging only
status	lapi_status_t	OUT - future support

LAPI_RESERVE_DGSP

You can use this operation to reserve a DGSP. This operation is provided because a LAPI client might cache a LAPI DGSP handle for later use. The client needs to ensure the DGSP will not be discarded before the cached handle is used. A DGSP handle, which is defined to be a pointer to a DGSP description that is already cached inside LAPI, is passed to this operation. The DGSP handle is also defined to be a

structure pointer, so that client programs can get direct access to information in the DGSP. Unless the client can be certain that the DGSP will not be "unreserved" by another thread while it is being accessed, the client should bracket the access window with its own reserve/unreserve operation. The client is not to modify the cached DGSP, but LAPI has no way to enforce this. The reserve operation increments the user reference count, thus protecting the DGSP until an unreserve operation occurs. This is needed because the thread that placed the reservation will expect to be able to use or examine the cached DGSP until it makes an unreserve call (which decrements the user reference count), even if the unreserve operation that matches the original register operation occurs within this window on some other thread.

LAPI_RESERVE_DGSP uses the `lapi_resv_dgsp_t` command structure.

Table 4. The lapi_resv_dgsp_t fields

lapi_resv_dgsp_t field	lapi_resv_dgsp_t field type	lapi_resv_dgsp_t usage
Util_type	`lapi_util_type_t`	LAPI_RESERVE_DGSP
dgsp_handle	`lapi_dg_handle_t`	OUT - handle for a registered DGSP program
in_usr_func	`lapi_usr_fcall_t`	For debugging only
status	`lapi_status_t`	OUT - future support

LAPI_UNRESERVE_DGSP

You can use this operation to unregister or unreserve a DGSP. This operation decrements the user reference count. If external and internal reference counts are zero, this operation lets LAPI free the DGSP. All operations that decrement a reference count cause LAPI to check to see if the counts have both become `0` and if they have, dispose of the DGSP. Several internal LAPI activities increment and decrement a second reference count. The cached DGSP is disposable only when all activities (internal and external) that depend on it and use reference counting to preserve it have discharged their reference. The DGSP handle is passed to LAPI as a value parameter and LAPI does not nullify the caller's handle. It is your responsibility to not use this handle again because in doing an unreserve operation, you have indicated that you no longer count on the handle remaining valid.

LAPI_UNRESERVE_DGSP uses the `lapi_dref_dgsp_t` command structure.

Table 5. The lapi_dref_dgsp_t fields

lapi_dref_dgsp_t field	lapi_dref_dgsp_t field type	lapi_dref_dgsp_t usage
Util_type	`lapi_util_type_t`	LAPI_UNRESERVE_DGSP
dgsp_handle	`lapi_dg_handle_t`	OUT - handle for a registered DGSP program
in_usr_func	`lapi_usr_fcall_t`	For debugging only
status	`lapi_status_t`	OUT - future support

LAPI_REG_DDM_FUNC

You can use this operation to register data distribution manager (DDM) functions. It works in conjunction with the DGSM CONTROL instruction. Primarily, it is used for `MPI_Accumulate`, but LAPI clients can provide any DDM function. It is also used to establish a callback function for processing data that is being scattered into a user buffer on the destination side.

The native LAPI user can install a callback without affecting the one MPI has registered for `MPI_Accumulate`. The function prototype for the callback function is:

```
typedef long ddm_func_t (        /* return number of bytes processed */
        void      *in,           /* pointer to inbound data          */
        void      *inout,        /* pointer to destination space     */
        long      bytes,         /* number of bytes inbound          */
        int       operand,       /* CONTROL operand value            */
```

```
        int       operation      /* CONTROL operation value          */
);
```

A DDM function acts between the arrival of message data and the target buffer. The most common usage is to combine inbound data with data already in the target buffer. For example, if the target buffer is an array of integers and the incoming message consists of integers, the DDM function can be written to add each incoming integer to the value that is already in the buffer. The *operand* and *operation* fields of the DDM function allow one DDM function to support a range of operations with the CONTROL instruction by providing the appropriate values for these fields.

See *RSCT for AIX 5L: LAPI Programming Guide* for more information about DGSP programming.

LAPI_REG_DDM_FUNC uses the `lapi_reg_ddm_t` command structure. Each call replaces the previous function pointer, if there was one.

Table 6. The lapi_reg_ddm_t fields

lapi_reg_ddm_t field	lapi_reg_ddm_t field type	lapi_reg_ddm_t usage
Util_type	`lapi_util_type_t`	LAPI_REG_DDM_FUNC
ddm_func	`ddm_func_t *`	IN - DDM function pointer
in_usr_func	`lapi_usr_fcall_t`	For debugging only
status	`lapi_status_t`	OUT - future support

LAPI_DGSP_PACK

You can use this operation to gather data to a pack buffer from a user buffer under control of a DGSP. A single buffer may be packed by a series of calls. The caller provides a *position* value that is initialized to the starting offset within the buffer. Each pack operation adjusts *position*, so the next pack operation can begin where the previous pack operation ended. In general, a series of pack operations begins with *position* initialized to `0`, but any offset is valid. There is no state carried from one pack operation to the next. Each pack operation starts at the beginning of the DGSP it is passed.

LAPI_DGSP_PACK uses the `lapi_pack_dgsp_t` command structure.

Table 7. The lapi_pack_dgsp_t fields

lapi_pack_dgsp_t field	lapi_pack_dgsp_t field type	lapi_pack_dgsp_t usage
Util_type	`lapi_util_type_t`	LAPI_DGSP_PACK
dgsp_handle	`lapi_dg_handle_t`	OUT - handle for a registered DGSP program
in_buf	`void *`	IN - source buffer to pack
bytes	`ulong`	IN - number of bytes to pack
out_buf	`void *`	OUT - output buffer for pack
out_size	`ulong`	IN - output buffer size in bytes
position	`ulong`	IN/OUT - current buffer offset
in_usr_func	`lapi_usr_fcall_t`	For debugging only
status	`lapi_status_t`	OUT - future support

LAPI_DGSP_UNPACK

You can use this operation to scatter data from a packed buffer to a user buffer under control of a DGSP. A single buffer may be unpacked by a series of calls. The caller provides a *position* value that is initialized to the starting offset within the packed buffer. Each unpack operation adjusts *position*, so the

next unpack operation can begin where the previous unpack operation ended. In general, a series of unpack operations begins with *position* initialized to 0, but any offset is valid. There is no state carried from one unpack operation to the next. Each unpack operation starts at the beginning of the DGSP it is passed.

LAPI_DGSP_UNPACK uses the lapi_unpack_dgsp_t command structure.

Table 8. The lapi_unpack_dgsp_t fields

lapi_unpack_dgsp_t field	lapi_unpack_dgsp_t field type	lapi_unpack_dgsp_t usage
Util_type	lapi_util_type_t	LAPI_DGSP_UNPACK
dgsp_handle	lapi_dg_handle_t	OUT - handle for a registered DGSP program
buf	void *	IN - source buffer for unpack
in_size	ulong	IN - source buffer size in bytes
out_buf	void *	OUT - output buffer for unpack
bytes	ulong	IN - number of bytes to unpack
out_size	ulong	IN - output buffer size in bytes
position	ulong	IN/OUT - current buffer offset
in_usr_func	lapi_usr_fcall_t	For debugging only
status	lapi_status_t	OUT - future support

LAPI_ADD_UDP_DEST_PORT

You can use this operation to update UDP port information about the destination task. This operation can be used when you have written your own UDP handler (*udp_hndlr*) and you need to support recovery of failed tasks. You cannot use this operation under the POE runtime environment.

LAPI_ADD_UDP_DEST_PORT uses the lapi_add_udp_port_t command structure.

Table 9. The lapi_add_udp_port_t fields

lapi_add_udp_port_t field	lapi_add_udp_port_t field type	lapi_add_udp_port_t usage
Util_type	lapi_util_type_t	LAPI_ADD_UDP_DEST_PORT
tgt	uint	IN - destination task ID
udp_port	lapi_udp_t *	IN - UDP port information for the target
instance_no	uint	IN - Instance number of UDP
in_usr_func	lapi_usr_fcall_t	For debugging only
status	lapi_status_t	OUT - future support

LAPI_GET_THREAD_FUNC

You can use this operation to retrieve various shared locking and signaling functions. Retrieval of these functions is valid only after LAPI is initialized and before LAPI is terminated. You should not call any of these functions after LAPI is terminated.

LAPI_GET_THREAD_FUNC uses the lapi_thread_func_t command structure.

Table 10. The lapi_thread_func_t fields		
lapi_thread_func_t field	**lapi_thread_func_t field type**	**lapi_thread_func_t usage**
Util_type	`lapi_util_type_t`	`LAPI_GET_THREAD_FUNC`
mutex_lock	`lapi_mutex_lock_t`	OUT - mutex lock function pointer
mutex_unlock	`lapi_mutex_unlock_t`	OUT - mutex unlock function pointer
mutex_trylock	`lapi_mutex_trylock_t`	OUT - mutex try lock function pointer
mutex_getowner	`lapi_mutex_getowner_t`	OUT - mutex get owner function pointer
cond_wait	`lapi_cond_wait_t`	OUT - condition wait function pointer
cond_timedwait	`lapi_cond_timedwait_t`	OUT - condition timed wait function pointer
cond_signal	`lapi_cond_signal_t`	OUT - condition signal function pointer
cond_init	`lapi_cond_init_t`	OUT - initialize condition function pointer
cond_destroy	`lapi_cond_destroy_t`	OUT - destroy condition function pointer

LAPI uses the pthread library for thread ID management. You can therefore use `pthread_self()` to get the running thread ID and `lapi_mutex_getowner_t` to get the thread ID that owns the shared lock. Then, you can use `pthread_equal()` to see if the two are the same.

Mutex thread functions

`LAPI_GET_THREAD_FUNC` includes the following mutex thread functions: mutex lock, mutex unlock, mutex try lock, and mutex get owner.

`Mutex lock function pointer`

```
int (*lapi_mutex_lock_t)(lapi_handle_t hndl);
```

This function acquires the lock that is associated with the specified LAPI handle. The call blocks if the lock is already held by another thread. Deadlock can occur if the calling thread is already holding the lock. You are responsible for preventing and detecting deadlocks.

`Parameters`

INPUT

hndl
Specifies the LAPI handle.

`Return values`

0
Indicates that the lock was acquired successfully.

EINVAL
Is returned if the lock is not valid because of an incorrect *hndl* value.

`Mutex unlock function pointer`

```
int (*lapi_mutex_unlock_t)(lapi_handle_t hndl);
```

This function releases the lock that is associated with the specified LAPI handle. A thread should only unlock its own locks.

`Parameters`

INPUT

hndl
Specifies the LAPI handle.

Return values

0
Indicates that the lock was released successfully.

EINVAL
Is returned if the lock is not valid because of an incorrect *hndl* value.

Mutex try lock function pointer

```
int (*lapi_mutex_trylock_t)(lapi_handle_t hndl);
```

This function tries to acquire the lock that is associated with the specified LAPI handle, but returns immediately if the lock is already held.

Parameters

INPUT

hndl
Specifies the LAPI handle.

Return values

0
Indicates that the lock was acquired successfully.

EBUSY
Indicates that the lock is being held.

EINVAL
Is returned if the lock is not valid because of an incorrect *hndl* value.

Mutex get owner function pointer

```
int (*lapi_mutex_getowner_t)(lapi_handle_t hndl, pthread_t *tid);
```

This function gets the pthread ID of the thread that is currently holding the lock associated with the specified LAPI handle. LAPI_NULL_THREAD_ID indicates that the lock is not held at the time the function is called.

Parameters

INPUT

hndl
Specifies the LAPI handle.

OUTPUT

tid
Is a pointer to hold the pthread ID to be retrieved.

Return values

0
Indicates that the lock owner was retrieved successfully.

EINVAL
Is returned if the lock is not valid because of an incorrect *hndl* value.

Condition functions

LAPI_GET_THREAD_FUNC includes the following condition functions: condition wait, condition timed wait, condition signal, initialize condition, and destroy condition.

Condition wait function pointer

```
int (*lapi_cond_wait_t)(lapi_handle_t hndl, lapi_cond_t *cond);
```

This function waits on a condition variable (*cond*). The user must hold the lock associated with the LAPI handle (*hndl*) before making the call. Upon the return of the call, LAPI guarantees that the lock is still

being held. The same LAPI handle must be supplied to concurrent `lapi_cond_wait_t` operations on the same condition variable.

`Parameters`

INPUT

hndl
Specifies the LAPI handle.

cond
Is a pointer to the condition variable to be waited on.

`Return values`

0
Indicates that the condition variable has been signaled.

EINVAL
Indicates that the value specified by *hndl* or *cond* is not valid.

`Condition timed wait function pointer`

```
int (*lapi_cond_timedwait_t)(lapi_handle_t hndl,
                             lapi_cond_t *cond,
                             struct timespec *timeout);
```

This function waits on a condition variable (*cond*). The user must hold the lock associated with the LAPI handle (*hndl*) before making the call. Upon the return of the call, LAPI guarantees that the lock is still being held. The same LAPI handle must be supplied to concurrent `lapi_cond_timedwait_t` operations on the same condition variable.

`Parameters`

INPUT

hndl
Specifies the LAPI handle.

cond
Is a pointer to the condition variable to be waited on.

timeout
Is a pointer to the absolute time structure specifying the timeout.

`Return values`

0
Indicates that the condition variable has been signaled.

ETIMEDOUT
Indicates that time specified by *timeout* has passed.

EINVAL
Indicates that the value specified by *hndl*, *cond*, or *timeout* is not valid.

`Condition signal function pointer`

```
int (*lapi_cond_wait_t)(lapi_handle_t hndl, lapi_cond_t *cond);
typedef int (*lapi_cond_signal_t)(lapi_handle_t hndl, lapi_cond_t *cond);
```

This function signals a condition variable (*cond*) to wake up a thread that is blocked on the condition. If there are multiple threads waiting on the condition variable, which thread to wake up is decided randomly.

`Parameters`

INPUT

hndl
Specifies the LAPI handle.

cond
Is a pointer to the condition variable to be signaled.

Return values

0
Indicates that the condition variable has been signaled.

EINVAL
Indicates that the value specified by *hndl* or *cond* is not valid.

Initialize condition function pointer

```
int (*lapi_cond_init_t)(lapi_handle_t hndl, lapi_cond_t *cond);
```

This function initializes a condition variable.

Parameters

INPUT

hndl
Specifies the LAPI handle.

cond
Is a pointer to the condition variable to be initialized.

Return values

0
Indicates that the condition variable was initialized successfully.

EAGAIN
Indicates that the system lacked the necessary resources (other than memory) to initialize another condition variable.

ENOMEM
Indicates that there is not enough memory to initialize the condition variable.

EINVAL
Is returned if the *hndl* value is not valid.

Destroy condition function pointer

```
int (*lapi_cond_destroy_t)(lapi_handle_t hndl, lapi_cond_t *cond);
```

This function destroys a condition variable.

Parameters

INPUT

hndl
Specifies the LAPI handle.

cond
Is a pointer to the condition variable to be destroyed.

Return values

0
Indicates that the condition variable was destroyed successfully.

EBUSY
Indicates that the implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced (while being used in a `lapi_cond_wait_t` or `lapi_cond_timedwait_t` by another thread, for example).

EINVAL
Indicates that the value specified by *hndl* or *cond* is not valid.

Parameters

INPUT

hndl
Specifies the LAPI handle.

INPUT/OUTPUT

util_cmd
Specifies the command type of the utility function.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_DGSP
Indicates that the DGSP that was passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) or is not a registered DGSP.

LAPI_ERR_DGSP_ATOM
Indicates that the DGSP has an *atom_size* that is less than 0 or greater than MAX_ATOM_SIZE.

LAPI_ERR_DGSP_BRANCH
Indicates that the DGSP attempted a branch that fell outside of the code array. This is returned only in validation mode.

LAPI_ERR_DGSP_COPY_SZ
Is returned with DGSP validation turned on when MCOPY block < 0 or COPY instruction with bytes < 0. This is returned only in validation mode.

LAPI_ERR_DGSP_FREE
Indicates that LAPI tried to free a DGSP that is not valid or is no longer registered. There should be one LAPI_UNRESERVE_DGSP operation to close the LAPI_REGISTER_DGSP operation and one LAPI_UNRESERVE_DGSP operation for each LAPI_RESERVE_DGSP operation.

LAPI_ERR_DGSP_OPC
Indicates that the DGSP *opcode* is not valid. This is returned only in validation mode.

LAPI_ERR_DGSP_STACK
Indicates that the DGSP has a greater GOSUB depth than the allocated stack supports. Stack allocation is specified by the dgsp->depth member. This is returned only in validation mode.

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_MEMORY_EXHAUSTED
Indicates that LAPI is unable to obtain memory from the system.

LAPI_ERR_UDP_PORT_INFO
Indicates that the *udp_port* information pointer is NULL (in C) or that the value of *udp_port* is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_UTIL_CMD
Indicates that the command type is not valid.

C Examples

1. To create and register a DGSP:

```
{
      /*
      ** DGSP code array.  DGSP instructions are stored
      ** as ints (with constants defined in lapi.h for
      ** the number of integers needed to store each
      ** instruction).  We will have one COPY and one ITERATE
      ** instruction in our DGSP.  We use LAPI's constants
      ** to allocate the appropriate storage.
```

```
*/
int code[LAPI_DGSM_COPY_SIZE+LAPI_DGSM_ITERATE_SIZE];

/* DGSP description */
lapi_dgsp_descr_t dgsp_d;

/*
** Data structure for the xfer call.
*/
lapi_xfer_t   xfer_struct;


/* DGSP data structures */
lapi_dgsm_copy_t     *copy_p;   /* copy instruction    */
lapi_dgsm_iterate_t  *iter_p;   /* iterate instruction */
int                  *code_ptr; /* code pointer        */

/* constant for holding code array info */
int                   code_less_iterate_size;

/* used for DGSP registration */
lapi_reg_dgsp_t     reg_util;


/*
** Set up dgsp description
*/

/* set pointer to code array */
dgsp_d.code = &code[0];

/* set size of code array */
dgsp_d.code_size = LAPI_DGSM_COPY_SIZE + LAPI_DGSM_ITERATE_SIZE;

/* not using DGSP gosub instruction */
dgsp_d.depth = 1;

/*
** set density to show internal gaps in the
** DGSP data layout
*/
dgsp_d.density = LAPI_DGSM_SPARSE;

/* transfer 4 bytes at a time */
dgsp_d.size = 4;

/* advance the template by 8 for each iteration */
dgsp_d.extent = 8;

/*
** ext specifies the memory 'footprint' of
** data to be transferred. The lext specifies
** the offset from the base address to begin
** viewing the data.  The rext specifies the
** length from the base address to use.
*/
dgsp_d.lext = 0;
dgsp_d.rext = 4;
/* atom size of 0 lets LAPI choose the packet size */
dgsp_d.atom_size = 0;

/*
** set up the copy instruction
*/
copy_p = (lapi_dgsm_copy_t *)(dgsp_d.code);
copy_p->opcode = LAPI_DGSM_COPY;

/* copy 4 bytes at a time */
copy_p->bytes = (long) 4;

/* start at offset 0 */
copy_p->offset = (long) 0;

/* set code pointer to address of iterate instruction */
code_less_iterate_size =  dgsp_d.code_size - LAPI_DGSM_ITERATE_SIZE;
code_ptr = ((int *)(code))+code_less_iterate_size;

/*
** Set up iterate instruction
*/
iter_p = (lapi_dgsm_iterate_t *) code_ptr;
iter_p->opcode = LAPI_DGSM_ITERATE;
```

```
        iter_p->iter_loc = (-code_less_iterate_size);

        /* Set up and do DGSP registration */
        reg_util.Util_type = LAPI_REGISTER_DGSP;
        reg_util.idgsp = &dgsp_d;
        LAPI_Util(hndl, (lapi_util_t *)&reg_util);

        /*
        ** LAPI returns a usable DGSP handle in
        ** reg_util.dgsp_handle
        ** Use this handle for subsequent reserve/unreserve
        ** and Xfer calls.  On the receive side, this
        ** handle can be returned by the header handler using the
        ** return_info_t mechanism.  The DGSP will then be used for
        ** scattering data.
        */

}
```

2. To reserve a DGSP handle:

```
{

        reg_util.dgsp_handle = dgsp_handle;

        /*
        ** dgsp_handle has already been created and
        ** registered as in the above example
        */

        reg_util.Util_type = LAPI_RESERVE_DGSP;
        LAPI_Util(hndl, (lapi_util_t *)&reg_util);

        /*
        ** LAPI's internal reference count to dgsp_handle
        ** will be incremented.  DGSP will
        ** remain available until an unreserve is
        ** done for each reserve, plus one more for
        ** the original registration.
        */

}
```

3. To unreserve a DGSP handle:

```
{

        reg_util.dgsp_handle = dgsp_handle;

        /*
        ** dgsp_handle has already created and
        ** registered as in the above example, and
        ** this thread no longer needs it.
        */

        reg_util.Util_type = LAPI_UNRESERVE_DGSP;
        LAPI_Util(hndl, (lapi_util_t *)&reg_util);

        /*
        ** An unreserve is required for each reserve,
        ** plus one more for the original registration.
        */

}
```

Location

/usr/lib/liblapi_r.a

Related Information

Subroutines: LAPI_Init, LAPI_Xfer

LAPI_Waitcntr Subroutine

Purpose

Waits until a specified counter reaches the value specified.

Library

Availability Library (`liblapi_r.a`)

C Syntax

```
#include <lapi.h>

int LAPI_Waitcntr(hndl, cntr, val, cur_cntr_val)
lapi_handle_t  hndl;
lapi_cntr_t   *cntr;
int            val;
int           *cur_cntr_val;
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_WAITCNTR(hndl, cntr, val, cur_cntr_val, ierror)
INTEGER hndl
TYPE (LAPI_CNTR_T) :: cntr
INTEGER val
INTEGER cur_cntr_val
INTEGER ierror
```

Description

`Type of call:` local progress monitor (blocking)

This subroutine waits until *cntr* reaches or exceeds the specified *val*. Once *cntr* reaches *val*, *cntr* is decremented by the value of *val*. In this case, "decremented" is used (as opposed to "set to zero") because *cntr* could have contained a value that was greater than the specified *val* when the call was made. This call may or may not check for message arrivals over the LAPI context *hndl*. The *cur_cntr_val* variable is set to the current counter value.

Parameters

INPUT

hndl
Specifies the LAPI handle.

val
Specifies the value the counter needs to reach.

INPUT/OUTPUT

cntr
Specifies the counter structure (in FORTRAN) to be waited on or its address (in C). The value of this parameter cannot be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

OUTPUT

cur_cntr_val
Specifies the integer value of the current counter. This value can be NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Restrictions

LAPI statistics are *not* reported for shared memory communication and data transfer, or for messages that a task sends to itself.

C Examples

To wait on a counter to reach a specified value:

```
{

      int          val;
      int          cur_cntr_val;
      lapi_cntr_t  some_cntr;
      .
      .
      .
      LAPI_Waitcntr(hndl, &some_cntr, val, &cur_cntr_val);
      /* Upon return, some_cntr has reached val */

}
```

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_CNTR_NULL
Indicates that the *cntr* pointer is NULL (in C) or that the value of *cntr* is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

Location

/usr/lib/liblapi_r.a

LAPI_Xfer Subroutine

Purpose

Serves as a wrapper function for LAPI data transfer functions.

Library

Availability Library (liblapi_r.a)

C Syntax

```
#include <lapi.h>

int LAPI_Xfer(hndl, xfer_cmd)
lapi_handle_t hndl;
lapi_xfer_t *xfer_cmd;


typedef struct {
    uint            src;          /* Target task ID    */
    uint            reason;       /* LAPI return codes */
    ulong           reserve[6];   /* Reserved          */
} lapi_sh_info_t;


typedef void (scompl_hndlr_t)(lapi_handle_t *hndl, void *completion_param,
```

```
                                lapi_sh_info_t *info);
```

FORTRAN Syntax

```
include 'lapif.h'

LAPI_XFER(hndl, xfer_cmd, ierror)
INTEGER hndl
TYPE (fortran_xfer_type) :: xfer_cmd
INTEGER ierror
```

Description

`Type of call:` point-to-point communication (non-blocking)

The `LAPI_Xfer` subroutine provides a superset of the functionality of these subroutines: `LAPI_Amsend`, `LAPI_Amsendv`, `LAPI_Put`, `LAPI_Putv`, `LAPI_Get`, `LAPI_Getv`, and `LAPI_Rmw`. In addition, `LAPI_Xfer` provides data gather/scatter program (DGSP) messages transfer.

In C, the `LAPI_Xfer` command is passed a pointer to a union. It examines the first member of the union, `Xfer_type`, to determine the transfer type, and to determine which union member was passed. `LAPI_Xfer` expects every field of the identified union member to be set. It does not examine or modify any memory outside of the identified union member. `LAPI_Xfer` treats all union members (except `status`) as read-only data.

This subroutine provides the following functions:

- The remote address fields are expanded to be of type `lapi_long_t`, which is long enough for a 64-bit address. This allows a 32-bit task to send data to 64-bit addresses, which may be important in client/server programs.
- `LAPI_Xfer` allows the origin counter to be replaced with a send completion callback.
- `LAPI_Xfer` is used to transfer data using LAPI's data gather/scatter program (DGSP) interface.

The `lapi_xfer_t` structure is defined as:

```
typedef union {
        lapi_xfer_type_t   Xfer_type;
        lapi_get_t         Get;
        lapi_am_t          Am;
        lapi_rmw_t         Rmw;
        lapi_put_t         Put;
        lapi_getv_t        Getv;
        lapi_putv_t        Putv;
        lapi_amv_t         Amv;
        lapi_amdgsp_t      Dgsp;
} lapi_xfer_t;
```

Though the `lapi_xfer_t` structure applies only to the C version of `LAPI_Xfer`, the following tables include the FORTRAN equivalents of the C datatypes.

Table 11 on page 803 list the values of the `lapi_xfer_type_t` structure for C and the explicit *Xfer_type* values for FORTRAN.

Table 11. LAPI_Xfer structure types

Value of *Xfer_type* (C or FORTRAN)	**Union member as interpreted by LAPI_Xfer (C)**	**Value of *fortran_xfer_type* (FORTRAN)**
LAPI_AM_XFER	lapi_am_t	LAPI_AM_T
LAPI_AMV_XFER	lapi_amv_t	LAPI_AMV_T
LAPI_DGSP_XFER	lapi_amdgsp_t	LAPI_AMDGSP_T
LAPI_GET_XFER	lapi_get_t	LAPI_GET_T

Table 11. LAPI_Xfer structure types (continued)		
Value of *Xfer_type* (C or FORTRAN)	**Union member as interpreted by LAPI_Xfer (C)**	**Value of *fortran_xfer_type* (FORTRAN)**
LAPI_GETV_XFER	lapi_getv_t	LAPI_GETV_T
LAPI_PUT_XFER	lapi_put_t	LAPI_PUT_T
LAPI_PUTV_XFER	lapi_putv_t	LAPI_PUTV_T
LAPI_RMW_XFER	lapi_rmw_t	LAPI_RMW_T

lapi_am_t details

Table 12 on page 804 shows the correspondence among the parameters of the LAPI_Amsend subroutine, the fields of the C lapi_am_t structure and their datatypes, and the equivalent FORTRAN datatypes. The lapi_am_t fields are listed in Table 12 on page 804 in the order that they occur in the lapi_xfer_t structure.

Table 12. LAPI_Amsend and lapi_am_t equivalents			
lapi_am_t field name (C)	**lapi_am_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Amsend parameter**
Xfer_type	lapi_xfer_type_t	INTEGER(KIND = 4)	implicit in C LAPI_Xfer value in FORTRAN: LAPI_AM_XFER
flags	int	INTEGER(KIND = 4)	none LAPI_Xfer parameter in FORTRAN: *flags*
tgt	uint	INTEGER(KIND = 4)	*tgt*
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN: *pad*
hdr_hdl	lapi_long_t	INTEGER(KIND = 8)	*hdr_hdl*
uhdr_len	uint	INTEGER(KIND = 4)	*uhdr_len*
none	none	INTEGER(KIND = 4)	LAPI_Xfer parameter in FORTRAN (64-bit): *pad2*
uhdr	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	*uhdr*
udata	void *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	*udata*
udata_len	ulong	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	*udata_len*
shdlr	scompl_hndlr_t *	INTEGER(KIND = 4) (32-bit) INTEGER(KIND = 8) (64-bit)	none LAPI_Xfer parameter in FORTRAN: *shdlr*

Table 12. LAPI_Amsend and lapi_am_t equivalents (continued)			
lapi_am_t field name (C)	**lapi_am_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Amsend parameter**
sinfo	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *sinfo*
tgt_cntr	`lapi_long_t`	`INTEGER(KIND = 8)`	*tgt_cntr*
org_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_cntr*
cmpl_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*cmpl_cntr*

When the origin data buffer is free to be used, the pointer to the send completion handler (*shdlr*) is called with the send completion data (*sinfo*) if *shdlr* is not a NULL pointer (in C) or `LAPI_ADDR_NULL` (in FORTRAN). Otherwise, the behavior is identical to that of `LAPI_Amsend`.

lapi_amv_t details

Table 13 on page 805 shows the correspondence among the parameters of the `LAPI_Amsendv` subroutine, the fields of the C `lapi_amv_t` structure and their datatypes, and the equivalent FORTRAN datatypes. The `lapi_amv_t` fields are listed in Table 13 on page 805 in the order that they occur in the `lapi_xfer_t` structure.

Table 13. LAPI_Amsendv and lapi_amv_t equivalents			
lapi_amv_t field name (C)	**lapi_amv_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Amsendv parameter**
Xfer_type	`lapi_xfer_type_t`	`INTEGER(KIND = 4)`	implicit in C `LAPI_Xfer` value in FORTRAN: `LAPI_AMV_XFER`
flags	`int`	`INTEGER(KIND = 4)`	none `LAPI_Xfer` parameter in FORTRAN: *flags*
tgt	`uint`	`INTEGER(KIND = 4)`	*tgt*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN: *pad*
hdr_hdl	`lapi_long_t`	`INTEGER(KIND = 8)`	*hdr_hdl*
uhdr_len	`uint`	`INTEGER(KIND = 4)`	*uhdr_len*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN (64-bit): *pad2*
uhdr	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*uhdr*
shdlr	`scompl_hndlr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *shdlr*

Table 13. LAPI_Amsendv and lapi_amv_t equivalents (continued)			
lapi_amv_t field name (C)	**lapi_amv_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Amsendv parameter**
sinfo	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *sinfo*
org_vec	lapi_vec_t *	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_vec*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN (32-bit): *pad2*
tgt_cntr	`lapi_long_t`	`INTEGER(KIND = 8)`	*tgt_cntr*
org_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_cntr*
cmpl_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*cmpl_cntr*

lapi_amdgsp_t details

Table 14 on page 806 shows the correspondence among the fields of the C `lapi_amdgsp_t` structure and their datatypes, how they are used in `LAPI_Xfer`, and the equivalent FORTRAN datatypes. The `lapi_amdgsp_t` fields are listed in Table 14 on page 806 in the order that they occur in the `lapi_xfer_t` structure.

Table 14. The lapi_amdgsp_t fields			
lapi_amdgsp_t field name (C)	**lapi_amdgsp_t field type (C)**	**Equivalent FORTRAN datatype**	**LAPI_Xfer usage**
Xfer_type	`lapi_xfer_type_t`	`INTEGER(KIND = 4)`	`LAPI_DGSP_XFER`
flags	`int`	`INTEGER(KIND = 4)`	This field allows users to specify directives or hints to LAPI. If you do not want to use any directives or hints, you must set this field to `0`. See The lapi_amdgsp_t flags field for more information.
tgt	`uint`	`INTEGER(KIND = 4)`	target task
none	none	`INTEGER(KIND = 4)`	*pad* (padding alignment for FORTRAN only)
hdr_hdl	`lapi_long_t`	`INTEGER(KIND = 8)`	header handler to invoke at target
uhdr_len	`uint`	`INTEGER(KIND = 4)`	user header length (multiple of processor's doubleword size)
none	none	`INTEGER(KIND = 4)`	*pad2* (padding alignment for 64-bit FORTRAN only)

Table 14. The lapi_amdgsp_t fields (continued)			
lapi_amdgsp_t field name (C)	**lapi_amdgsp_t field type (C)**	**Equivalent FORTRAN datatype**	**LAPI_Xfer usage**
uhdr	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	pointer to user header
udata	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	pointer to user data
udata_len	`ulong`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	user data length
shdlr	`scompl_hndlr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	send completion handler (optional)
sinfo	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	data pointer to pass to send completion handler (optional)
tgt_cntr	`lapi_long_t`	`INTEGER(KIND = 8)`	target counter (optional)
org_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	origin counter (optional)
cmpl_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	completion counter (optional)
dgsp	`lapi_dg_handle_t`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	Handle of a registered DGSP
status	`lapi_status_t`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	Status to return (future use)
none	none	`INTEGER(KIND = 4)`	*pad3* (padding alignment for 64-bit FORTRAN only)

When the origin data buffer is free to be modified, the send completion handler (*shdlr*) is called with the send completion data (*sinfo*), if *shdlr* is not a NULL pointer (in C) or `LAPI_ADDR_NULL` (in FORTRAN).

See Using lapi_am_dgsp_t for scatter-side DGSP processing for more information.

The lapi_amdgsp_t flags field

One or more flags can be set using the | (bitwise or) operator. User directives are always followed and could result in incorrect results if used improperly. Appropriate hints might improve performance, but they may be ignored by LAPI. Inappropriate hints might degrade performance, but they will not cause incorrect results.

The following directive flags are defined:

USE_TGT_VEC_TYPE
Instructs LAPI to use the vector type of the target vector (*tgt_vec*). In other words, *tgt_vec* is to be interpreted as type `lapi_vec_t`; otherwise, it is interpreted as type `lapi_lvec_t`. The `lapi_lvec_t` type uses `lapi_long_t`. The `lapi_vec_t` type uses `void *` or `long`. Incorrect results will occur if one type is used in place of the other.

BUFFER_BOTH_CONTIGUOUS
Instructs LAPI to treat all data to be transferred as contiguous, which can improve performance. If this flag is set when non-contiguous data is sent, data will likely be corrupted.

The following hint flags are defined:

LAPI_NOT_USE_BULK_XFER
Instructs LAPI not to use bulk transfer, independent of the current setting for the task.

LAPI_USE_BULK_XFER
Instructs LAPI to use bulk transfer, independent of the current setting for the task.

If neither of these hint flags is set, LAPI will use the behavior defined for the task. If both of these hint flags are set, LAPI_NOT_USE_BULK_XFER will take precedence.

These hints may or may not be honored by the communication library.

Using lapi_am_dgsp_t for scatter-side DGSP processing

LAPI allows additional information to be returned from the header handler through the use of the `lapi_return_info_t` datatype. See *RSCT for AIX 5L: LAPI Programming Guide* for more information about `lapi_return_info_t`. In the case of transfer type `lapi_am_dgsp_t`, this mechanism can be used to instruct LAPI to run a user DGSP to scatter data on the receive side.

To use this mechanism, pass a `lapi_return_info_t *` pointer back to LAPI through the *msg_len* member of the user header handler. The *dgsp_handle* member of the passed structure must point to a DGSP description that has been registered on the receive side. See `LAPI_Util` and *RSCT for AIX 5L: LAPI Programming Guide* for details on building and registering DGSPs.

lapi_get_t details

Table 15 on page 808 shows the correspondence among the parameters of the `LAPI_Get` subroutine, the fields of the C `lapi_get_t` structure and their datatypes, and the equivalent FORTRAN datatypes. The `lapi_get_t` fields are listed in Table 15 on page 808 in the order that they occur in the `lapi_xfer_t` structure.

Table 15. LAPI_Get and lapi_get_t equivalents

lapi_get_t field name (C)	lapi_get_t field type (C)	Equivalent FORTRAN datatype	Equivalent LAPI_Get parameter
Xfer_type	`lapi_xfer_type_t`	`INTEGER(KIND = 4)`	implicit in C `LAPI_Xfer` value in FORTRAN: `LAPI_GET_XFER`
flags	`int`	`INTEGER(KIND = 4)`	none `LAPI_Xfer` parameter in FORTRAN: *flags*
tgt	`uint`	`INTEGER(KIND = 4)`	*tgt*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN: *pad*
tgt_addr	`lapi_long_t`	`INTEGER(KIND = 8)`	*tgt_addr*
org_addr	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_addr*
len	`ulong`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*len*
tgt_cntr	`lapi_long_t`	`INTEGER(KIND = 8)`	*tgt_cntr*

Table 15. LAPI_Get and lapi_get_t equivalents (continued)			
lapi_get_t field name (C)	**lapi_get_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Get parameter**
org_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_cntr*
chndlr	`compl_hndlr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *chndlr*
cinfo	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *cinfo*

When the origin data buffer has completely arrived, the pointer to the completion handler (*chndlr*) is called with the completion data (*cinfo*), if *chndlr* is not a NULL pointer (in C) or `LAPI_ADDR_NULL` (in FORTRAN). Otherwise, the behavior is identical to that of `LAPI_Get`.

lapi_getv_t details

Table 16 on page 809 shows the correspondence among the parameters of the `LAPI_Getv` subroutine, the fields of the C `lapi_getv_t` structure and their datatypes, and the equivalent FORTRAN datatypes. The `lapi_getv_t` fields are listed in Table 15 on page 808 in the order that they occur in the `lapi_xfer_t` structure.

Table 16. LAPI_Getv and lapi_getv_t equivalents			
lapi_getv_t field name (C)	**lapi_getv_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Getv parameter**
Xfer_type	`lapi_xfer_type_t`	`INTEGER(KIND = 4)`	implicit in C `LAPI_Xfer` value in FORTRAN: `LAPI_GETV_XFER`
flags	`int`	`INTEGER(KIND = 4)`	none `LAPI_Xfer` parameter in FORTRAN: *flags*
tgt	`uint`	`INTEGER(KIND = 4)`	*tgt*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN (64-bit): *pad*
org_vec	`lapi_vec_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_vec*
tgt_vec	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*tgt_vec*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN (32-bit): *pad*
tgt_cntr	`lapi_long_t`	`INTEGER(KIND = 8)`	*tgt_cntr*
org_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_cntr*

Table 16. LAPI_Getv and lapi_getv_t equivalents (continued)			
lapi_getv_t field name (C)	**lapi_getv_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Getv parameter**
chndlr	`compl_hndlr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *chndlr*
cinfo	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *cinfo*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN (32-bit): *pad2*

The `flags` field accepts `USE_TGT_VEC_TYPE` (see lapi_amdgsp_t flags field) to indicate that `tgt_vec` is to be interpreted as type `lapi_vec_t`; otherwise, it is interpreted as type `lapi_lvec_t`. Note the corresponding field is `lapi_vec_t` in the `LAPI_Getv` call.

When the origin data buffer has completely arrived, the pointer to the completion handler (`chndlr`) is called with the completion data (`cinfo`) if `chndlr` is not a NULL pointer (in C) or `LAPI_ADDR_NULL` (in FORTRAN). Otherwise, the behavior is identical to that of `LAPI_Getv`.

lapi_put_t details

Table 17 on page 810 shows the correspondence among the parameters of the `LAPI_Put` subroutine, the fields of the C `lapi_put_t` structure and their datatypes, and the equivalent FORTRAN datatypes. The `lapi_put_t` fields are listed in Table 17 on page 810 in the order that they occur in the `lapi_xfer_t` structure.

Table 17. LAPI_Put and lapi_put_t equivalents			
lapi_put_t field name (C)	**lapi_put_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Put parameter**
Xfer_type	`lapi_xfer_type_t`	`INTEGER(KIND = 4)`	implicit in C `LAPI_Xfer` value in FORTRAN: `LAPI_PUT_XFER`
flags	`int`	`INTEGER(KIND = 4)`	none `LAPI_Xfer` parameter in FORTRAN: *flags*
tgt	`uint`	`INTEGER(KIND = 4)`	*tgt*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN: *pad*
tgt_addr	`lapi_long_t`	`INTEGER(KIND = 8)`	*tgt_addr*
org_addr	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_addr*
len	`ulong`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*len*

Table 17. LAPI_Put and lapi_put_t equivalents (continued)			
lapi_put_t field name (C)	**lapi_put_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Put parameter**
shdlr	`scompl_hndlr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *shdlr*
sinfo	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *sinfo*
tgt_cntr	`lapi_long_t`	`INTEGER(KIND = 8)`	*tgt_cntr*
org_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_cntr*
cmpl_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*cmpl_cntr*

When the origin data buffer is free to be used, the pointer to the send completion handler (*shdlr*) is called with the send completion data (*sinfo*), if *shdlr* is not a NULL pointer (in C) or `LAPI_ADDR_NULL` (in FORTRAN). Otherwise, the behavior is identical to that of `LAPI_Put`.

lapi_putv_t details

Table 18 on page 811 shows the correspondence among the parameters of the `LAPI_Putv` subroutine, the fields of the C `lapi_putv_t` structure and their datatypes, and the equivalent FORTRAN datatypes. The `lapi_putv_t` fields are listed in Table 17 on page 810 in the order that they occur in the `lapi_xfer_t` structure.

Table 18. LAPI_Putv and lapi_putv_t equivalents			
lapi_putv_t field name (C)	**lapi_putv_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Putv parameter**
Xfer_type	`lapi_xfer_type_t`	`INTEGER(KIND = 4)`	implicit in C `LAPI_Xfer` value in FORTRAN: `LAPI_PUT_XFER`
flags	`int`	`INTEGER(KIND = 4)`	none `LAPI_Xfer` parameter in FORTRAN: *flags*
tgt	`uint`	`INTEGER(KIND = 4)`	*tgt*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN (64-bit): *pad*
shdlr	`scompl_hndlr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *shdlr*
sinfo	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *sinfo*

Table 18. LAPI_Putv and lapi_putv_t equivalents (continued)			
lapi_putv_t field name (C)	**lapi_putv_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Putv parameter**
org_vec	`lapi_vec_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_vec*
tgt_vec	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*tgt_vec*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN (32-bit): *pad*
tgt_cntr	`lapi_long_t`	`INTEGER(KIND = 8)`	*tgt_cntr*
org_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_cntr*
cmpl_cntr	`lapi_cntr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*cmpl_cntr*

The `flags` field accepts `USE_TGT_VEC_TYPE` (see lapi_amdgsp_t flags field) to indicate that `tgt_vec` is to be interpreted as `lapi_vec_t`; otherwise, it is interpreted as a `lapi_lvec_t`. Note that the corresponding field is `lapi_vec_t` in the `LAPI_Putv` call.

When the origin data buffer is free to be modified, the pointer to the send completion handler (`shdlr`) is called with the send completion data (`sinfo`), if `shdlr` is not a NULL pointer (in C) or `LAPI_ADDR_NULL` (in FORTRAN). Otherwise, the behavior is identical to that of `LAPI_Putv`.

lapi_rmw_t details

Table 19 on page 812 shows the correspondence among the parameters of the `LAPI_Rmw` subroutine, the fields of the C `lapi_rmw_t` structure and their datatypes, and the equivalent FORTRAN datatypes. The `lapi_rmw_t` fields are listed in Table 17 on page 810 in the order that they occur in the `lapi_xfer_t` structure.

Table 19. LAPI_Rmw and lapi_rmw_t equivalents			
lapi_rmw_t field name (C)	**lapi_rmw_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Rmw parameter**
Xfer_type	`lapi_xfer_type_t`	`INTEGER(KIND = 4)`	implicit in C `LAPI_Xfer` value in FORTRAN: `LAPI_RMW_XFER`
op	`Rmw_ops_t`	`INTEGER(KIND = 4)`	*op*
tgt	`uint`	`INTEGER(KIND = 4)`	*tgt*
size	`uint`	`INTEGER(KIND = 4)`	implicit in C `LAPI_Xfer` parameter in FORTRAN: *size* (must be 32 or 64)
tgt_var	lapi_long_t	`INTEGER(KIND = 8)`	*tgt_var*
in_val	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*in_val*

Table 19. LAPI_Rmw and lapi_rmw_t equivalents (continued)			
lapi_rmw_t field name (C)	**lapi_rmw_t field type (C)**	**Equivalent FORTRAN datatype**	**Equivalent LAPI_Rmw parameter**
prev_tgt_val	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*prev_tgt_val*
org_cntr	`lapi_cntr t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	*org_cntr*
shdlr	`scompl_hndlr_t *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *shdlr*
sinfo	`void *`	`INTEGER(KIND = 4)` (32-bit) `INTEGER(KIND = 8)` (64-bit)	none `LAPI_Xfer` parameter in FORTRAN: *shdlr*
none	none	`INTEGER(KIND = 4)`	`LAPI_Xfer` parameter in FORTRAN (32-bit): *pad*

When the origin data buffer is free to be used, the pointer to the send completion handler (*shdlr*) is called with the send completion data (*sinfo*), if *shdlr* is not a NULL pointer (in C) or `LAPI_ADDR_NULL` (in FORTRAN). The *size* value must be either 32 or 64, indicating whether you want the *in_val* and *prev_tgt_val* fields to point to a 32-bit or 64-bit quantity, respectively. Otherwise, the behavior is identical to that of `LAPI_Rmw`.

Parameters

INPUT

hndl
Specifies the LAPI handle.

xfer_cmd
Specifies the name and parameters of the data transfer function.

OUTPUT

ierror
Specifies a FORTRAN return code. This is always the last parameter.

Return Values

LAPI_SUCCESS
Indicates that the function call completed successfully.

LAPI_ERR_DATA_LEN
Indicates that the value of *udata_len* or *len* is greater than the value of LAPI constant `LAPI_MAX_MSG_SZ`.

LAPI_ERR_DGSP
Indicates that the DGSP that was passed in is NULL (in C) or `LAPI_ADDR_NULL` (in FORTRAN) or is not a registered DGSP.

LAPI_ERR_DGSP_ATOM
Indicates that the DGSP has an *atom_size* that is less than `0` or greater than MAX_ATOM_SIZE.

LAPI_ERR_DGSP_BRANCH
Indicates that the DGSP attempted a branch that fell outside the code array.

LAPI_ERR_DGSP_CTL
Indicates that a DGSP control instruction was encountered in a non-valid context (such as a gather-side control or scatter-side control with an atom size of 0 at gather, for example).

LAPI_ERR_DGSP_OPC
Indicates that the DGSP op-code is not valid.

LAPI_ERR_DGSP_STACK
Indicates that the DGSP has greater GOSUB depth than the allocated stack supports. Stack allocation is specified by the dgsp->depth member.

LAPI_ERR_HDR_HNDLR_NULL
Indicates that the *hdr_hdl* passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_HNDL_INVALID
Indicates that the *hndl* passed in is not valid (not initialized or in terminated state).

LAPI_ERR_IN_VAL_NULL
Indicates that the *in_val* pointer is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_MEMORY_EXHAUSTED
LAPI is unable to obtain memory from the system.

LAPI_ERR_OP_SZ
Indicates that the lapi_rmw_t *size* field is not set to 32 or 64.

LAPI_ERR_ORG_ADDR_NULL
Indicates either that the *udata* parameter passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and *udata_len* is greater than 0, or that the *org_addr* passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and *len* is greater than 0.

Note: if *Xfer_type* = LAPI_DGSP_XFER, the case in which *udata* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN) and *udata_len* is greater than 0 is valid, so an error is not returned.

LAPI_ERR_ORG_EXTENT
Indicates that the *org_vec*'s extent (stride * *num_vecs*) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_STRIDE
Indicates that the *org_vec* stride is less than block.

LAPI_ERR_ORG_VEC_ADDR
Indicates that the *org_vec->info[i]* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (*org_vec->len[i]*) is not 0.

LAPI_ERR_ORG_VEC_LEN
Indicates that the sum of *org_vec*->len is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_ORG_VEC_NULL
Indicates that the *org_vec* value is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_ORG_VEC_TYPE
Indicates that the *org_vec->vec_type* is not valid.

LAPI_ERR_RMW_OP
Indicates the op is not valid.

LAPI_ERR_STRIDE_ORG_VEC_ADDR_NULL
Indicates that the strided vector address *org_vec->info*[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_STRIDE_TGT_VEC_ADDR_NULL
Indicates that the strided vector address *tgt_vec->info*[0] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT
Indicates that the *tgt* passed in is outside the range of tasks defined in the job.

LAPI_ERR_TGT_ADDR_NULL
Indicates that *ret_addr* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_EXTENT
Indicates that *tgt_vec*'s extent (stride * *num_vecs*) is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_PURGED
Indicates that the subroutine returned early because LAPI_Purge_totask() was called.

LAPI_ERR_TGT_STRIDE
Indicates that the *tgt_vec* stride is less than block.

LAPI_ERR_TGT_VAR_NULL
Indicates that the *tgt_var* address is NULL (in C) or that the value of *tgt_var* is LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_VEC_ADDR
Indicates that the *tgt_vec*->*info*[i] is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but its length (*tgt_vec*->*len*[i]) is not 0.

LAPI_ERR_TGT_VEC_LEN
Indicates that the sum of *tgt_vec*->*len* is greater than the value of LAPI constant LAPI_MAX_MSG_SZ.

LAPI_ERR_TGT_VEC_NULL
Indicates that *tgt_vec* is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN).

LAPI_ERR_TGT_VEC_TYPE
Indicates that the *tgt_vec*->*vec_type* is not valid.

LAPI_ERR_UHDR_LEN
Indicates that the *uhdr_len* value passed in is greater than MAX_UHDR_SZ or is not a multiple of the processor's doubleword size.

LAPI_ERR_UHDR_NULL
Indicates that the *uhdr* passed in is NULL (in C) or LAPI_ADDR_NULL (in FORTRAN), but *uhdr_len* is not 0.

LAPI_ERR_VEC_LEN_DIFF
Indicates that *org_vec* and *tgt_vec* have different lengths (*len[ ]*).

LAPI_ERR_VEC_NUM_DIFF
Indicates that *org_vec* and *tgt_vec* have different *num_vecs*.

LAPI_ERR_VEC_TYPE_DIFF
Indicates that *org_vec* and *tgt_vec* have different vector types (*vec_type*).

LAPI_ERR_XFER_CMD
Indicates that the *Xfer_cmd* is not valid.

C Examples

1. To run the sample code shown in LAPI_Get using the LAPI_Xfer interface:

```
{

      lapi_xfer_t xfer_struct;

      /* initialize the table buffer for the data addresses */

      /* get remote data buffer addresses */
      LAPI_Address_init(hndl,(void *)data_buffer,data_buffer_list);
      .
      .
      .
      /* retrieve data_len bytes from address data_buffer_list[tgt] on */
      /* task tgt. write the data starting at address data_buffer.     */
      /* tgt_cntr and org_cntr can be NULL.                            */

      xfer_struct.Get.Xfer_type = LAPI_GET_XFER;
      xfer_struct.Get.flags = 0;
      xfer_struct.Get.tgt = tgt;
      xfer_struct.Get.tgt_addr = data_buffer_list[tgt];
      xfer_struct.Get.org_addr = data_buffer;
      xfer_struct.Get.len = data_len;
```

```
      xfer_struct.Get.tgt_cntr = tgt_cntr;
      xfer_struct.Get.org_cntr = org_cntr;

      LAPI_Xfer(hndl, &xfer_struct);

}
```

2. To implement the LAPI_STRIDED_VECTOR example from LAPI_Amsendv using the LAPI_Xfer interface:

```
{
    lapi_xfer_t   xfer_struct;                        /* info for LAPI_Xfer call
*/
    lapi_vec_t    vec;                                /* data for data transfer
*/
    .
    .
    .
    vec->num_vecs = NUM_VECS;        /* NUM_VECS = number of vectors to transfer
*/
                                     /* must match that of the target vector
*/
    vec->vec_type = LAPI_GEN_STRIDED_XFER;              /* same as target vector
*/


    vec->info[0]  = buffer_address;    /* starting address for data copy
*/
    vec->info[1]  = block_size;        /* bytes of data to copy
*/
    vec->info[2]  = stride;            /* distance from copy block to copy block
*/
    /* data will be copied as follows:
*/
    /* block_size bytes will be copied from buffer_address
*/
    /* block_size bytes will be copied from buffer_address+stride
*/
    /* block_size bytes will be copied from buffer_address+(2*stride)
*/
    /* block_size bytes will be copied from buffer_address+(3*stride)
*/
    .
    .
    .
    /* block_size bytes will be copied from buffer_address+((NUM_VECS-1)*stride)
*/
    .
    .
    .
    xfer_struct.Amv.Xfer_type = LAPI_AMV_XFER;
    xfer_struct.Amv.flags     = 0;
    xfer_struct.Amv.tgt       = tgt;
    xfer_struct.Amv.hdr_hdl   = hdr_hdl_list[tgt];
    xfer_struct.Amv.uhdr_len  = uhdr_len;  /* user header length */
    xfer_struct.Amv.uhdr      = uhdr;


    /* LAPI_AMV_XFER allows the use of a send completion handler */
    /* If non-null, the shdlr function is invoked at the point   */
    /* the origin counter would increment.  Note that both the   */
    /* org_cntr and shdlr can be used.                           */
    /* The user's shdlr must be of type scompl_hndlr_t *.        */
    /* scompl_hndlr_t is defined in /usr/include/lapi.h          */
    xfer_struct.shdlr = shdlr;

    /* Use sinfo to pass user-defined data into the send         */
    /* completion handler, if desired.                           */
    xfer_struct.sinfo     = sinfo;       /* send completion data */

    xfer_struct.org_vec   = vec;
```

```
    xfer_struct.tgt_cntr  = tgt_cntr;
    xfer_struct.org_cntr  = org_cntr;
    xfer_struct.cmpl_cntr = cmpl_cntr;


    LAPI_Xfer(hndl, &xfer_struct);
    .
    .
    .
}
```

See the LAPI_Amsendv subroutine for more information about the header handler definition.

Location

/usr/lib/liblapi_r.a

layout_object_create Subroutine

Purpose

Initializes a layout context.

Library

Layout Library (**libi18n.a**)

Syntax

```
#include <sys/lc_layout.h>
```

```
int layout_object_create (locale_name, layout_object)
const char * locale_name;
LayoutObject * layout_object;
```

Description

The **layout_object_create** subroutine creates the **LayoutObject** structure associated with the locale specified by the *locale_name* parameter. The **LayoutObject** structure is a symbolic link containing all the data and methods necessary to perform the layout operations on context dependent and bidirectional characters of the locale specified.

When the **layout_object_create** subroutine completes without errors, the *layout_object* parameter points to a valid **LayoutObject** structure that can be used by other BIDI subroutines. The returned **LayoutObject** structure is initialized to an initial state that defines the behavior of the BIDI subroutines. This initial state is locale dependent and is described by the layout values returned by the **layout_object_getvalue** subroutine. You can change the layout values of the **LayoutObject** structure using the **layout_object_setvalue** subroutine. Any state maintained by the **LayoutObject** structure is independent of the current global locale set with the **setlocale** subroutine.

Note: If you are developing internationalized applications that may support multibyte locales, please see **Use of the libcur Package** in *General Programming Concepts: Writing and Debugging Programs*

Parameters

Item	Description
locale_name	Specifies a locale. It is recommended that you use the **LC_CTYPE** category by calling the **setlocale** (**LC_CTYPE**,NULL) subroutine.

Item	Description
layout_object	Points to a valid **LayoutObject** structure that can be used by other layout subroutines. This parameter is used only when the **layout_object_create** subroutine completes without errors. The *layout_object* parameter is not set and a non-zero value is returned if a valid **LayoutObject** structure cannot be created.

Return Values

Upon successful completion, the **layout_object_create** subroutine returns a value of 0. The *layout_object* parameter points to a valid handle.

Error Codes

If the **layout_object_create** subroutine fails, it returns the following error codes:

Item	Description
LAYOUT_EINVAL	The locale specified by the *locale_name* parameter is not available.
LAYOUT_EMFILE	The OPEN_MAX value of files descriptors are currently open in the calling process.
LAYOUT_ENOMEM	Insufficient storage space is available.

layout_object_editshape or wcslayout_object_editshape Subroutine

Purpose

Edits the shape of the context text.

Library

Layout library (**libi18n.a**)

Syntax

```
#include <sys/lc_layout.h>
```

```
int layout_editshape ( layout_object, EditType, index, InpBuf, Inpsize,
OutBuf, OutSize)
LayoutObject layout_object;
BooleanValue EditType;
size_t *index;
const char *InpBuf;
size_t *Inpsize;
void *OutBuf;
size_t *OutSize;
```

```
int wcslayout_object_editshape(layout_object, EditType, index, InpBuf, Inpsize, OutBuf, OutSize)
LayoutObject layout_object;
BooleanValue EditType;
size_t *index;
const wchar_t *InpBuf;
size_t *InpSize;
```

```
void *OutBuf;
size_t *OutSize;
```

Description

The **layout_object_editshape** and **wcslayout_object_editshape** subroutines provide the shapes of the context text. The shapes are defined by the code element specified by the *index* parameter and any surrounding code elements specified by the ShapeContextSize layout value of the **LayoutObject** structure. The *layout_object* parameter specifies this **LayoutObject** structure.

Use the **layout_object_editshape** subroutine when editing code elements of one byte. Use the **wcslayout_object_editshape** subroutine when editing single code elements of multibytes. These subroutines do not affect any state maintained by the **layout_object_transform** or **wcslayout_object_transform** subroutine.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *General Programming Concepts: Writing and Debugging Programs*

Parameters

Item	Description
layout_object	Specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.
EditType	Specifies the type of edit shaping. When the *EditType* parameter stipulates the `EditInput` field, the subroutine reads the current code element defined by the *index* parameter and any preceding code elements defined by ShapeContextSize layout value of the **LayoutObject** structure. When the *EditType* parameter stipulates the `EditReplace` field, the subroutine reads the current code element defined by the *index* parameter and any surrounding code elements defined by ShapeContextSize layout value of the **LayoutObject** structure. **Note:** The editing direction defined by the Orientation and TEXT_VISUAL of the TypeOfText layout values of the **LayoutObject** structure determines which code elements are preceding and succeeding. When the ActiveShapeEditing layout value of the **LayoutObject** structure is set to True, the **LayoutObject** structure maintains the state of the `EditInput` field that may affect subsequent calls to these subroutines with the `EditInput` field defined by the *EditType* parameter. The state of the `EditInput` field of **LayoutObject** structure is not affected when the *EditType* parameter is set to the `EditReplace` field. To reset the state of the `EditInput` field to its initial state, call these subroutines with the *InpBuf* parameter set to NULL. The state of the `EditInput` field is not affected if errors occur within the subroutines.
index	Specifies an offset (in bytes) to the start of a code element in the *InpBuf* parameter on input. The *InpBuf* parameter provides the base text to be edited. In addition, the context of the surrounding code elements is considered where the minimum set of code elements needed for the specific context dependent script(s) is identified by the ShapeContextSize layout value. If the set of surrounding code elements defined by the *index*, *InpBuf*, and *InpSize* parameters is less than the size of front and back of the ShapeContextSize layout value, these subroutines assume there is no additional context available. The caller must provide the minimum context if it is available. The *index* parameter is in units associated with the type of the *InpBuf* parameter. On return, the *index* parameter is modified to indicate the offset to the first code element of the *InpBuf* parameter that required shaping. The number of code elements that required shaping is indicated on return by the *InpSize* parameter.

Item	Description
InpBuf	Specifies the source to be processed. A Null value with the `EditInput` field in the *EditType* parameter indicates a request to reset the state of the `EditInput` field to its initial state. Any portion of the *InpBuf* parameter indicates the necessity for redrawing or shaping.
InpSize	Specifies the number of code elements to be processed in units on input. These units are associated with the types for these subroutines. A value of -1 indicates that the input is delimited by a Null code element. On return, the value is modified to the actual number of code elements needed by the *InpBuf* parameter. A value of 0 when the value of the *EditType* parameter is the `EditInput` field indicates that the state of the `EditInput` field is reset to its initial state. If the *OutBuf* parameter is not NULL, the respective shaped code elements are written into the *OutBuf* parameter.
OutBuf	Contains the shaped output text. You can specify this parameter as a Null pointer to indicate that no transformed text is required. If Null, the subroutines return the *index* and *InpSize* parameters, which specify the amount of text required, to be redrawn. The encoding of the *OutBuf* parameter depends on the ShapeCharset layout value defined in *layout_object* parameter. If the ActiveShapeEditing layout value is set to False, the encoding of the *OutBuf* parameter is to be the same as the code set of the locale associated with the specified **LayoutObject** structure.
OutSize	Specifies the size of the output buffer on input in number of bytes. Only the code elements required to be shaped are written into the *OutBuf* parameter. The output buffer should be large enough to contain the shaped result; otherwise, only partial shaping is performed. If the ActiveShapeEditing layout value is set to True, the *OutBuf* parameter should be allocated to contain at least the number of code elements in the *InpBuf* parameter multiplied by the value of the ShapeCharsetSize layout value. On return, the *OutSize* parameter is modified to the actual number of bytes placed in the output buffer. When the *OutSize* parameter is specified as 0, the subroutines calculate the size of an output buffer large enough to contain the transformed text from the input buffer. The result will be returned in this field. The content of the buffers specifies by the *InpBuf* and *OutBuf* parameters, and the value of the *InpSize* parameter, remain unchanged.

Return Values

Upon successful completion, these subroutines return a value of 0. The *index* and *InpSize* parameters return the minimum set of code elements required to be redrawn.

Error Codes

If these subroutines fail, they return the following error codes:

Item	Description
LAYOUT_EILSEQ	Shaping stopped due to an input code element that cannot be shaped. The *index* parameter indicates the code element that caused the error. This code element is either a valid code element that cannot be shaped according to the ShapeCharset layout value or an invalid code element not defined by the code set defined in the **LayoutObject** structure. Use the **mbtowc** or **wctomb** subroutine in the same locale as the **LayoutObject** structure to determine if the code element is valid.
LAYOUT_E2BIG	The output buffer is too small and the source text was not processed. The *index* and *InpSize* parameters are not guaranteed on return.
LAYOUT_EINVAL	Shaping stopped due to an incomplete code element or shift sequence at the end of input buffer. The *InpSize* parameter indicates the number of code elements successfully transformed. **Note:** You can use this error code to determine the code element causing the error.
LAYOUT_ERANGE	Either the *index* parameter is outside the range as defined by the *InpSize* parameter, more than 15 embedding levels are in the source text, or the *InpBuf* parameter contains unbalanced Directional Format (Push/Pop).

layout_object_getvalue Subroutine

Purpose

Queries the current layout values of a **LayoutObject** structure.

Library

Layout Library (**libi18n.a**)

Syntax

```
#include <sys/lc_layout.h>

int layout_object_getvalue( layout_object,  values,  index)
LayoutObject layout_object;
LayoutValues values;
int *index;
```

Description

The **layout_object_getvalue** subroutine queries the current setting of layout values within the **LayoutObject** structure. The *layout_object* parameter specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.

The name field of the LayoutValues structure contains the name of the layout value to be queried. The value field is a pointer to where the layout value is stored. The values are queried from the **LayoutObject** structure and represent its current state.

For example, if the layout value to be queried is of type T, the *value* parameter must be of type T*. If T itself is a pointer, the **layout_object_getvalue** subroutine allocates space to store the actual data. The caller must free this data by calling the **free(**T**)** subroutine with the returned pointer.

When setting the value field, an extra level of indirection is present that is not present using the **layout_object_setvalue** parameter. When you set a layout value of type T, the value field contains T. However, when querying the same layout value, the value field contains &T.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in*General Programming Concepts: Writing and Debugging Programs*

Parameters

Item	Description
layout_object	Specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.
values	Specifies an array of layout values of type LayoutValueRec that are to be queried in the **LayoutObject** structure. The end of the array is indicated by `name=0`.
index	Specifies a layout value to be queried. If the value cannot be queried, the *index* parameter causing the error is returned and the subroutine returns a non-zero value.

Return Values

Upon successful completion, the **layout_object_getvalue** subroutine returns a value of 0. All layout values were successfully queried.

Error Codes

If the **layout_object_getvalue** subroutine fails, it returns the following values:

Item	Description
LAYOUT_EINVAL	The layout value specified by the *index* parameter is unknown or the *layout_object* parameter is invalid.
LAYOUT_EMOMEM	Insufficient storage space is available.

Examples

The following example queries whether the locale is bidirectional and gets the values of the `in` and `out` orienations.

```
#include <sys/lc_layout.h>
#include <locale.h>
main()
{
LayoutObject plh;
int RC=0;
LayoutValues layout;
LayoutTextDescriptor Descr;
int index;

RC=layout_object_create(setlocale(LC_CTYPE,""),&plh); /* create object */
if (RC) {printf("Create error !!\n"); exit(0);}

layout=malloc(3*sizeof(LayoutValueRec));
                                        /* allocate layout array */
layout[0].name=ActiveBidirection;       /* set name */
layout[1].name=Orientation;             /* set name */
layout[1].value=(caddr_t)&Descr;
            /* send address of memory to be allocated by function */

layout[2].name=0;                     /* indicate end of array */
RC=layout_object_getvalue(plh,layout,&index);
if (RC) {printf("Getvalue error at %d !!\n",index); exit(0);}
printf("ActiveBidirection = %d \n",*(layout[0].value));
                                                  /*print output*/
printf("Orientation in = %x  out = %x \n", Descr->>in, Descr->>out);
```

```
free(layout);                          /* free layout array */
free (Descr);              /* free memory allocated by function */
RC=layout_object_free(plh);          /* free layout object */
if (RC) printf("Free error !!\n");
}
```

layout_object_setvalue Subroutine

Purpose

Sets the layout values of a **LayoutObject** structure.

Library

Layout Library (**libi18n.a**)

Syntax

```
#include <sys/lc_layout.h>
```

```
int layout_object_setvalue( layout_object,  values,  index)
LayoutObject layout_object;
LayoutValues values;
int *index;
```

Description

The **layout_object_setvalue** subroutine changes the current layout values of the **LayoutObject** structure. The *layout_object* parameter specifies the **LayoutObject** structure created by the **layout_object_create** subroutine. The values are written into the **LayoutObject** structure and may affect the behavior of subsequent layout functions.

Note: Some layout values do alter internal states maintained by a **LayoutObject** structure.

The name field of the LayoutValueRec structure contains the name of the layout value to be set. The value field contains the actual value to be set. The value field is large enough to support all types of layout values. For more information on layout value types, see **Layout Values for the Layout Library** in *General Programming Concepts: Writing and Debugging Programs* .

Note: If you are developing internationalized applications that may support multibyte locales, please see **Use of the libcur Package** in *General Programming Concepts: Writing and Debugging Programs*

Parameters

Item	Description
layout_object	Specifies the **LayoutObject** structure returned by the **layout_object_create** subroutine.
values	Specifies an array of layout values of the type LayoutValueRec that this subroutine sets. The end of the array is indicated by `name=0`.
index	Specifies a layout value to be queried. If the value cannot be queried, the index parameter causing the error is returned and the subroutine returns a non-zero value. If an error is generated, a subset of the values may have been previously set.

Return Values

Upon successful completion, the **layout_object_setvalue** subroutine returns a value of 0. All layout values were successfully set.

Error Codes

If the **layout_object_setvalue** subroutine fails, it returns the following values:

Item	Description
LAYOUT_EINVAL	The layout value specified by the *index* parameter is unknown, its value is invalid, or the *layout_object* parameter is invalid.
LAYOUT_EMFILE	The **(OPEN_MAX)** file descriptors are currently open in the calling process.
LAYOUT_ENOMEM	Insufficient storage space is available.

Examples

The following example sets the TypeofText value to Implicit and the out value to Visual.

```
#include <sys/lc_layout.h>
#include <locale.h>

main()
{
LayoutObject plh;
int RC=0;
LayoutValues layout;
LayoutTextDescriptor Descr;
int index;

RC=layout_object_create(setlocale(LC_CTYPE,""),&plh); /* create object */
if (RC) {printf("Create error !!\n"); exit(0);}

layout=malloc(2*sizeof(LayoutValueRec)); /*allocate layout array*/
Descr=malloc(sizeof(LayoutTextDescriptorRec)); /* allocate text descriptor */
layout[0].name=TypeOfText;            /* set name */
layout[0].value=(caddr_t)Descr;       /* set value */
layout[1].name=0;                     /* indicate end of array */

Descr->in=TEXT_IMPLICIT;
Descr->out=TEXT_VISUAL;  RC=layout_object_setvalue(plh,layout,&index);
if (RC) printf("SetValue error at %d!!\n",index); /* check return code */
free(layout);                        /* free allocated memory */
free (Descr);
RC=layout_object_free(plh);          /* free layout object */
if (RC) printf("Free error !!\n");
}
```

layout_object_shapeboxchars Subroutine

Purpose

Shapes box characters.

Library

Layout Library (**libi18n.a**)

Syntax

```
#include <sys/lc_layout.h>int layout_object_shapeboxchars
(layout_object,InpBuf,InpSize,OutBuf)
LayoutObject layout_object;
const char *InpBuf;
const size_t InpSize;
char *OutBuf;
```

Description

The **layout_object_shapeboxchars** subroutine shapes box characters into the VT100 box character set.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *General Programming Concepts: Writing and Debugging Programs*

Parameters

Item	Description
layout_object	Specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.
InpBuf	Specifies the source text to be processed.
InpSize	Specifies the number of code elements to be processed.
OutBuf	Contains the shaped output text.

Return Values

Upon successful completion, this subroutine returns a value of 0.

Error Codes

If this subroutine fails, it returns the following values:

Item	Description
LAYOUT_EILSEQ	Shaping stopped due to an input code element that cannot be mapped into the VT100 box character set.
LAYOUT_EINVAL	Shaping stopped due to an incomplete code element or shift sequence at the end of the input buffer.

layout_object_transform or wcslayout_object_transform Subroutine

Purpose

Transforms text according to the current layout values of a **LayoutObject** structure.

Library

Layout Library (**libi18n.a**)

Syntax

```
#include <sys/lc_layout.h>
int layout_object_transform
( layout_object,  InpBuf,  InpSize,  OutBuf,  OutSize,  InpToOut,  OutToInp,
BidiLvl)
LayoutObject layout_object;
const char *InpBuf;
size_t *InpSize;
void * OutBuf;
size_t *OutSize;
size_t  *InpToOut;
size_t *OutToInp;
unsigned char *BidiLvl;
```

```
int wcslayout_object_transform
(layout_object, InpBuf, InpSize, OutBuf, OutSize, InpToOut, OutToInp, BidiLvl)
LayoutObject layout_object;
const char *InpBuf;
size_t *InpSize;
void *OutBuf;
Size_t *OutSize;
size_t *InpToOut;
size_t *OutToInp;
unsigned char *BidiLvl;
```

Description

The **layout_object_transform** and **wcslayout_object_transform** subroutines transform the text specified by the *InpBuf* parameter according to the current layout values in the **LayoutObject** structure. Any layout value whose type is LayoutTextDescriptor describes the attributes within the *InpBuf* and *OutBuf* parameters. If the attributes are the same as the *InpBuf* and *OutBuf* parameters themselves, a null transformation is done with respect to that specific layout value.

The output of these subroutines may be one or more of the following results depending on the setting of the respective parameters:

Item	Description
OutBuf, *OutSize*	Any transformed data is stored in the *OutBuf* parameter.
InpToOut	A cross reference from each code element of the *InpBuf* parameter to the transformed data.
OutToInp	A cross reference to each code element of the *InpBuf* parameter from the transformed data.
BidiLvl	A weighted value that represents the directional level of each code element of the *InpBuf* parameter. The level is dependent on the internal directional algorithm of the **LayoutObject** structure.

You can specify each of these output parameters as Null to indicate that no output is needed for the specific parameter. However, you should set at least one of these parameters to a nonNULL value to perform any significant work.

To perform shaping of a text string without reordering of code elements, set the TypeOfText layout value to **TEXT_VISUAL** and the in and out values of the Orientation layout value alike. These layout values are in the **LayoutObject** structure.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *General Programming Concepts: Writing and Debugging Programs*

Parameters

Item	Description
layout_object	Specifies the **LayoutObject** structure created by the **layout_object_create** subroutine.
InpBuf	Specifies the source text to be processed. This parameter cannot be null.
InpSize	Specifies the units of code elements processed associated with the bytes for the **layout_object_transform** and **wcslayout_object_transform** subroutines. A value of -1 indicates that the input is delimited by a null code element. On return, the value is modified to the actual number of code elements processed in the *InBuf* parameter. However, if the value in the *OutSize* parameter is zero, the value of the *InpSize* parameter is not changed.

Item	Description
OutBuf	Contains the transformed data. You can specify this parameter as a null pointer to indicate that no transformed data is required. The encoding of the *OutBuf* parameter depends on the ShapeCharset layout value defined in the **LayoutObject** structure. If the ActiveShapeEditing layout value is set to True, the encoding of the *OutBuf* parameter is the same as the code set of the locale associated with the **LayoutObject** structure.
OutSize	Specifies the size of the output buffer in number of bytes. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the ActiveShapeEditing layout value is set to True, the *OutBuf* parameter should be allocated to contain at least the number of code elements multiplied by the ShapeCharsetSize layout value. On return, the *OutSize* parameter is modified to the actual number of bytes placed in this parameter. When you specify the *OutSize* parameter as 0, the subroutine calculates the size of an output buffer to be large enough to contain the transformed text. The result is returned in this field. The content of the buffers specified by the *InpBuf* and *OutBuf* parameters, and a value of the *InpSize* parameter remains unchanged.
InpToOut	Represents an array of values with the same number of code elements as the *InpBuf* parameter if *InpToOut* parameter is not a null pointer. On output, the *n*th value in *InpToOut* parameter corresponds to the *n*th code element in *InpBuf* parameter. This value is the index in *OutBuf* parameter which identifies the transformed ShapeCharset element of the *n*th code element in *InpBuf* parameter. You can specify the *InpToOut* parameter as null if no index array from the *InpBuf* to *OutBuf* parameters is desired.
OutTolnp	Represents an array of values with the same number of code elements as contained in the *OutBuf* parameter if the *OutToInp* parameter is not a null pointer. On output, the *n*th value in the *OutTolnp* parameter corresponds to the *n*th ShapeCharset element in the *OutBuf* parameter. This value is the index in the *InpBuf* parameter which identifies the original code element of the *n*th ShapeCharset element in the *OutBuf* parameter. You can specify the *OutTolnp* parameter as NULL if no index array from the *OutBuf* to *InpBuf* parameters is desired.
BidiLvl	Represents an array of values with the same number of elements as the source text if the *BidiLvl* parameter is not a null pointer. The *n*th value in the *BidiLvl* parameter corresponds to the *n*th code element in the *InpBuf* parameter. This value is the level of this code element as determined by the bidirectional algorithm. You can specify the *BidiLvl* parameter as null if a levels array is not desired.

Return Values

Upon successful completion, these subroutines return a value of 0.

Error Codes

If these subroutines fail, they return the following values:

Item	Description
LAYOUT_EILSEQ	Transformation stopped due to an input code element that cannot be shaped or is invalid. The *InpSize* parameter indicates the number of the code element successfully transformed. **Note:** You can use this error code to determine the code element causing the error. This code element is either a valid code element but cannot be shaped into the ShapeCharset layout value or is an invalid code element not defined by the code set of the locale of the **LayoutObject** structure. You can use the **mbtowc** and **wctomb** subroutines to determine if the code element is valid when used in the same locale as the **LayoutObject** structure.
LAYOUT_E2BIG	The output buffer is full and the source text is not entirely processed.
LAYOUT_EINVAL	Transformation stopped due to an incomplete code element or shift sequence at the end of the input buffer. The *InpSize* parameter indicates the number of the code elements successfully transformed. **Note:** You can use this error code to determine the code element causing the error.
LAYOUT_ERANGE	More than 15 embedding levels are in the source text or the *InpBuf* parameter contains unbalanced Directional Format (Push/Pop). When the size of *OutBuf* parameter is not large enough to contain the entire transformed text, the input text state at the end of the **LAYOUT_E2BIG** error code is returned. To resume the transformation on the remaining text, the application calls the **layout_object_transform** subroutine with the same **LayoutObject** structure, the same *InpBuf* parameter, and *InpSize* parameter set to 0.

Examples

The following is an example of transformation of both directional re-ordering and shaping.

Note:

1. Uppercase represent left-to-right characters; lowercase represent right-to-left characters.
2. xyz represent the shapes of cde.

```
Position:           0123456789
InpBuf:             AB cde 12Z

Position:           0123456789
OutBuf:             AB 12 zyxZ

Position:           0123456789
ToTarget:           0128765349

Position:           0123456789
ToSource:           0127865439

Position:           0123456789
BidiLevel:          0001111220
```

layout_object_free Subroutine

Purpose

Frees a **LayoutObject** structure.

Library

Layout library (**libi18n.a**)

Syntax

```
#include <sys/lc_layout.h>
```

int layout_object_free(*layout_object***)**
LayoutObject *layout_object*;

Description

The **layout_object_free** subroutine releases all the resources of the **LayoutObject** structure created by the **layout_object_create** subroutine. The *layout_object* parameter specifies this **LayoutObject** structure.

Note: If you are developing internationalized applications that may support multibyte locales, please see Use of the libcur Package in *General Programming Concepts: Writing and Debugging Programs*

Parameters

Item	Description
layout_object	Specifies a **LayoutObject** structure returned by the **layout_object_create** subroutine.

Return Values

Upon successful completion, the **layout_object_free** subroutine returns a value of 0. All resources associated with the *layout_object* parameter are successfully deallocated.

Error Codes

If the **layout_object_free** subroutine fails, it returns the following error code:

Item	Description
LAYOUT_EFAULT	Errors occurred while processing the request.

lckpwdf Subroutine

Purpose

Locks the password database file.

Library

Security Library (`libc.a`)

Syntax

```
#include <pwd.h>

int lckpwdf()
```

Description

The `lckpwdf` subroutine opens the temporary file and locks it to prevent the concurrent modification of the `/etc/passwd` and `/etc/security/passwd` database files.

The **ulckpwdf** subroutine can be called to release this lock. Both the **lckpwdf** and **ulckpwdf** subroutines use the `/etc/security/.pwdlck` database file as a lock file.

Note:

There is no protection against direct access of password database files or the programs that do not use the **lckpwdf** and **ulckpwdf** subroutines.

Return Values

Upon successful completion of attaining a lock, the **lckpwdf** subroutine returns a value of 0. Otherwise, a value of -1 is returned when the lock is acquired by other process.

ldahread Subroutine

Purpose

Reads the archive header of a member of an archive file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ar.h>
#include <ldfcn.h>

int ldahread(ldPointer,ArchiveHeader)
LDFILE *ldPointer;
ARCHDR *ArchiveHeader;
```

Description

If the **TYPE(***ldPointer***)** macro from the **ldfcn.h** file is the archive file magic number, the **ldahread** subroutine reads the archive header of the extended common object file currently associated with the *ldPointer* parameter into the area of memory beginning at the *ArchiveHeader* parameter.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.
ArchiveHeader	Points to a **ARCHDR** structure.

Return Values

The **ldahread** subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **ldahread** routine fails if the **TYPE(***ldPointer***)** macro does not represent an archive file, or if it cannot read the archive header.

ldclose or ldaclose Subroutine

Purpose

Closes a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldclose( ldPointer)
LDFILE *ldPointer;

int ldaclose(ldPointer)
LDFILE *ldPointer;
```

Description

The **ldopen** and **ldclose** subroutines provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of common object files can be processed as if it were a series of simple common object files.

If the **ldfcn.h** file **TYPE(***ldPointer***)** macro is the magic number of an archive file, and if there are any more files in the archive, the **ldclose** subroutine reinitializes the **ldfcn.h** file **OFFSET(***ldPointer***)** macro to the file address of the next archive member and returns a failure value. The **ldfile** structure is prepared for a subsequent **ldopen**.

If the **TYPE(***ldPointer***)** macro does not represent an archive file, the **ldclose** subroutine closes the file and frees the memory allocated to the **ldfile** structure associated with *ldPointer.*

The **ldaclose** subroutine closes the file and frees the memory allocated to the **ldfile** structure associated with the *ldPointer* parameter regardless of the value of the **TYPE**(*ldPointer***)** macro.

Parameters

Item	Description
ldPointer	Pointer to the **LDFILE** structure that was returned as the result of a successful call to **ldopen** or **ldaopen**.

Return Values

The **ldclose** subroutine returns a SUCCESS or FAILURE value.

The **ldaclose** subroutine always returns a SUCCESS value and is often used in conjunction with the **ldaopen** subroutine.

Error Codes

The **ldclose** subroutine returns a failure value if there are more files to archive.

ldexpd32, ldexpd64, and ldexpd128 Subroutines

Purpose

Loads the exponent of a decimal floating-point number.

Syntax

```
#include <math.h>

_Decimal32 ldexpd32 (x, exp)
_Decimal32 x;
int exp;

_Decimal64 ldexpd64 (x, exp)
_Decimal64 x;
int exp;

_Decimal128 ldexpd128 (x, exp)
_Decimal128 x;
int exp;
```

Description

The **ldexpd32**, **ldexpd64**, and **ldexpd128** subroutines compute the quantity *x* * 10^{exp}.

An application that wants to check for error situations must set the **errno** global variable to the value of zero and call the **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. On return, if the **errno** is of the value of nonzero or the **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is of the value of nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.
exp	Specifies the exponent of 10.

Return Values

Upon successful completion, the **ldexpd32**, **ldexpd64**, and **ldexpd128** subroutines return *x* multiplied by 10 to the power of *exp*.

If the **ldexpd32**, **ldexpd64**, or **ldexpd128** subroutines would cause overflow, a range error occurs and the **ldexpd32**, **ldexpd64**, and **ldexpd128** subroutines return ±**HUGE_VAL_D32**, ±**HUGE_VAL_D64**, and ±**HUGE_VAL_D128** (according to the sign of *x*), respectively.

If the correct value will cause underflow, and is not representable, a range error might occur, and 0.0 is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or Inf, *x* is returned.

If *exp* is 0, *x* is returned.

If the correct value will cause underflow, and is representable, a range error might occur and the correct value is returned.

ldexp, ldexpf, or ldexpl Subroutine

Purpose

Loads exponent of a floating-point number.

Syntax

```
#include <math.h>
float ldexpf (x, exp)
float x;
```

```
int exp;

long double ldexpl (x, exp)
long double x;
int exp;

double ldexp (x, exp)
double  x;
int exp;
```

Description

The **ldexpf**, **ldexpl**, and **ldexp** subroutines compute the quantity *x* * 2^{exp}.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.
exp	Specifies the exponent of 2.

Return Values

Upon successful completion, the **ldexpf**, **ldexpl**, and **ldexp** subroutines return *x* multiplied by 2, raised to the power *exp*.

If the **ldexpf**, **ldexpl**, or **ldexp** subroutines would cause overflow, a range error occurs and the **ldexpf**, **ldexpl**, and **ldexp** subroutines return ±**HUGE_VALF**, ±**HUGE_VALL**, and ±**HUGE_VAL** (according to the sign of *x*), respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or Inf, *x* is returned.

If *exp* is 0, *x* is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Error Codes

If the result of the **ldexp** or **ldexpl** subroutine overflows, then +/- **HUGE_VAL** is returned, and the global variable **errno** is set to **ERANGE**.

If the result of the **ldexp** or **ldexpl** subroutine underflows, 0 is returned, and the **errno** global variable is set to a **ERANGE** value.

ldfhread Subroutine

Purpose

Reads the file header of an XCOFF file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldfhread ( ldPointer,  FileHeader)
LDFILE *ldPointer;
void *FileHeader;
```

Description

The **ldfhread** subroutine reads the file header of the object file currently associated with the *ldPointer* parameter into the area of memory beginning at the *FileHeader* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the file header of the associated object file. Since the **ldopen** subroutine provides magic number information (via the **HEADER**(*ldPointer*)**.f_magic** macro), the calling application can always determine whether the *FileHeader* pointer should refer to a 32-bit FILHDR or 64-bit FILHDR_64 structure.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to **ldopen** or **ldaopen** subroutine.
FileHeader	Points to a buffer large enough to accommodate a **FILHDR** structure, according to the object mode of the file being read.

Return Values

The **ldfhread** subroutine returns Success or Failure.

Error Codes

The **ldfhread** subroutine fails if it cannot read the file header.

Note: In most cases, the use of **ldfhread** can be avoided by using the **HEADER** (*ldPointer*) macro defined in the **ldfcn.h** file. The information in any field or fieldname of the header file may be accessed using the **header** *(ldPointer*) **fieldname** macro.

Examples

The following is an example of code that opens an object file, determines its mode, and uses the **ldfhread** subroutine to acquire the file header. This code would be compiled with both **_XCOFF32_** and **_XCOFF64_** defined:

```
#define __XCOFF32__
#define __XCOFF64__

#include <ldfcn.h>


/* for each FileName to be processed */

if ( (ldPointer = ldopen(fileName, ldPointer)) != NULL)
{
    FILHDR    FileHead32;
    FILHDR_64 FileHead64;
    void      *FileHeader;
```

```
    if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
        FileHeader = &FileHead32;
    else if ( HEADER(ldPointer).f_magic == U803XTOCMAGIC )
        FileHeader = &FileHead64;
    else
        FileHeader = NULL;

    if ( FileHeader && (ldfhread( ldPointer, FileHeader ) == SUCCESS) )
    {
        /* ...successfully read header... */
        /* ...process according to magic number... */
    }
}
```

ldgetname Subroutine

Purpose

Retrieves symbol name for common object file symbol table entry.

Library

Object File Access Routine Library (**libld.a)**

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

char *ldgetname ( ldPointer,  Symbol)
LDFILE *ldPointer;
void *Symbol;
```

Description

The **ldgetname** subroutine returns a pointer to the name associated with *Symbol* as a string. The string is in a static buffer local to the **ldgetname** subroutine that is overwritten by each call to the **ldgetname** subroutine and must therefore be copied by the caller if the name is to be saved.

The common object file format handles arbitrary length symbol names with the addition of a string table. The **ldgetname** subroutine returns the symbol name associated with a symbol table entry for an XCOFF-format object file.

The calling routine to provide a pointer to a buffer large enough to contain a symbol table entry for the associated object file. Since the **ldopen** subroutine provides magic number information (via the **HEADER(***ldPointer***).f_magic** macro), the calling application can always determine whether the Symbol pointer should refer to a 32-bit SYMENT or 64-bit SYMENT_64 structure.

The maximum length of a symbol name is **BUFSIZ**, defined in the **stdio.h** file.

Parameters

Item	Description
ldPointer	Points to an **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.
Symbol	Points to an initialized 32-bit or 64-bit **SYMENT** structure.

Error Codes

The **ldgetname** subroutine returns a null value (defined in the **stdio.h** file) for a COFF-format object file if the name cannot be retrieved. This situation can occur if one of the following is true:

- The string table cannot be found.
- The string table appears invalid (for example, if an auxiliary entry is handed to the **ldgetname** subroutine wherein the name offset lies outside the boundaries of the string table).
- The name's offset into the string table is past the end of the string table.

Typically, the **ldgetname** subroutine is called immediately after a successful call to the **ldtbread** subroutine to retrieve the name associated with the symbol table entry filled by the **ldtbread** subroutine.

Examples

The following is an example of code that determines the object file type before making a call to the **ldtbread** and **ldgetname** subroutines.

```
#define __XCOFF32__
#define __XCOFF64__

#include <ldfcn.h>


SYMENT    Symbol32;
SYMENT_64 Symbol64;
void      *Symbol;

if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
    Symbol = &Symbol32;
else if ( HEADER(ldPointer).f_magic == U64_TOCMAGIC )
    Symbol = &Symbol64;
else
    Symbol = NULL;

if ( Symbol )
    /* for each symbol in the symbol table */
    for ( symnum = 0 ; symnum < HEADER(ldPointer).f_nsyms ; symnum++ )
    {
        if ( ldtbread(ldPointer,symnum,Symbol) == SUCCESS )
        {
            char *name = ldgetname(ldPointer,Symbol)

            if ( name )
            {
                /* Got the name... */
                .
                .
            }

            /* Increment symnum by the number of auxiliary entries */
            if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
                symnum += Symbol32.n_numaux;
            else if ( HEADER(ldPointer).f_magic == U64_TOCMAGIC )
                symnum += Symbol64.n_numaux;
        }
        else
        {
            /* Should have been a symbol...indicate the error */
                .
                .
        }
    }
```

ldlread, ldlinit, or ldlitem Subroutine

Purpose

Manipulates line number entries of a common object file function.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldlread ( ldPointer,  FunctionIndex,  LineNumber,  LineEntry)
LDFILE *ldPointer;
int FunctionIndex;
unsigned short LineNumber;
void *LineEntry;

int ldlinit (ldPointer, FunctionIndex)
LDFILE *ldPointer;
int FunctionIndex;

int ldlitem (ldPointer, LineNumber, LineEntry)
LDFILE *ldPointer;
unsigned short LineNumber;
void *LineEntry;
```

Description

The **ldlread** subroutine searches the line number entries of the XCOFF file currently associated with the *ldPointer* parameter. The **ldlread** subroutine begins its search with the line number entry for the beginning of a function and confines its search to the line numbers associated with a single function. The function is identified by the *FunctionIndex* parameter, the index of its entry in the object file symbol table. The **ldlread** subroutine reads the entry with the smallest line number equal to or greater than the *LineNumber* parameter into the memory beginning at the *LineEntry* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the line number entry for the associated object file type. Since the **ldopen** subroutine provides magic number information (via the **HEADER**(*ldPointer*)**.f_magic** macro), the calling application can always determine whether the *LineEntry* pointer should refer to a 32-bit LINENO or 64-bit LINENO_64 structure.

The **ldlinit** and **ldlitem** subroutines together perform the same function as the **ldlread** subroutine. After an initial call to the **ldlread** or **ldlinit** subroutine, the **ldlitem** subroutine may be used to retrieve successive line number entries associated with a single function. The **ldlinit** subroutine simply locates the line number entries for the function identified by the *FunctionIndex* parameter. The **ldlitem** subroutine finds and reads the entry with the smallest line number equal to or greater than the *LineNumber* parameter into the memory beginning at the *LineEntry* parameter.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** , **lddopen**,or **ldaopen** subroutine.
LineNumber	Specifies the index of the first *LineNumber* parameter entry to be read.
LineEntry	Points to a buffer that will be filled in with a **LINENO** structure from the object file.
FunctionIndex	Points to the symbol table index of a function.

Return Values

The **ldlread**, **ldlinit**, and **ldlitem** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldlread** subroutine fails if there are no line number entries in the object file, if the *FunctionIndex* parameter does not index a function entry in the symbol table, or if it finds no line number equal to or greater than the *LineNumber* parameter. The **ldlinit** subroutine fails if there are no line number entries in the object file or if the *FunctionIndex* parameter does not index a function entry in the symbol table. The **ldlitem** subroutine fails if it finds no line number equal to or greater than the *LineNumber* parameter.

ldlseek or ldnlseek Subroutine

Purpose

Seeks to line number entries of a section of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldlseek ( ldPointer,  SectionIndex)
LDFILE *ldPointer;
unsigned short SectionIndex;

int ldnlseek (ldPointer,  SectionName)
LDFILE *ldPointer;
char *SectionName;
```

Description

The **ldlseek** subroutine seeks to the line number entries of the section specified by the *SectionIndex* parameter of the common object file currently associated with the *ldPointer* parameter. The first section has an index of 1.

The **ldnlseek** subroutine seeks to the line number entries of the section specified by the *SectionName* parameter.

Both subroutines determine the object mode of the associated file before seeking to the relocation entries of the indicated section.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.
SectionIndex	Specifies the index of the section whose line number entries are to be seeked to.
SectionName	Specifies the name of the section whose line number entries are to be seeked to.

Return Values

The **ldlseek** and **ldnlseek** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldlseek** subroutine fails if the *SectionIndex* parameter is greater than the number of sections in the object file. The **ldnlseek** subroutine fails if there is no section name corresponding with the *SectionName* parameter. Either function fails if the specified section has no line number entries or if it cannot seek to the specified line number entries.

ldohseek Subroutine

Purpose

Seeks to the optional file header of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldohseek ( ldPointer)
LDFILE *ldPointer;
```

Description

The **ldohseek** subroutine seeks to the optional auxiliary header of the common object file currently associated with the *ldPointer* parameter. The subroutine determines the object mode of the associated file before seeking to the end of its file header.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to **ldopen** or **ldaopen** subroutine.

Return Values

The **ldohseek** subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **ldohseek** subroutine fails if the object file has no optional header, if the file is not a 32-bit or 64-bit object file, or if it cannot seek to the optional header.

ldopen or ldaopen Subroutine

Purpose

Opens an object or archive file for reading.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

LDFILE *ldopen( FileName,  ldPointer)
char *FileName;
LDFILE *ldPointer;

LDFILE *ldaopen(FileName, ldPointer)
char *FileName;
LDFILE *ldPointer;

LDFILE *lddopen(FileDescriptor, type, ldPointer)
int FileDescriptor;
char *type;
LDFILE *ldPointer;
```

Description

The **ldopen** and **ldclose** subroutines provide uniform access to both simple object files and object files that are members of archive files. Thus, an archive of object files can be processed as if it were a series of ordinary object files.

If the *ldPointer* is null, the **ldopen** subroutine opens the file named by the *FileName* parameter and allocates and initializes an **LDFILE** structure, and returns a pointer to the structure.

If the *ldPointer* parameter is not null and refers to an **LDFILE** for an archive, the structure is updated for reading the next archive member. In this case, and if the value of the **TYPE(***ldPointer***)** macro is the archive magic number **ARTYPE**.

The **ldopen** and **ldclose** subroutines are designed to work in concert. The **ldclose** subroutine returns failure only when the *ldPointer* refers to an archive containing additional members. Only then should the **ldopen** subroutine be called with a num-null *ldPointer* argument. In all other cases, in particular whenever a new *FileName* parameter is opened, the **ldopen** subroutine should be called with a null *ldPointer* argument.

If the value of the *ldPointer* parameter is not null, the **ldaopen** subroutine opens the *FileName* parameter again and allocates and initializes a new **LDFILE** structure, copying the **TYP**E, **OFFSET**, and **HEADER** fields from the *ldPointer* parameter. The **ldaopen** subroutine returns a pointer to the new **ldfile** structure. This new pointer is independent of the old pointer, *ldPointer*. The two pointers may be used concurrently to read separate parts of the object file. For example, one pointer may be used to step sequentially through the relocation information, while the other is used to read indexed symbol table entries.

The **lddopen** function accesses the previously opened file referenced by the *FileDescriptor* parameter. In all other respects, it functions the same as the **ldopen** subroutine.

The functions transparently open both 32-bit and 64-bit object files, as well as both small format and large format archive files. Once a file or archive is successfully opened, the calling application can examine the **HEADER(***ldPointer***).f_magic** field to check the magic number of the file or archive member associated with *ldPointer*. (This is necessary due to an archive potentially containing members that are not object files.) The magic numbers U802TOCMAGIC and U803XTOCMAGIC are defined in the **ldfcn.h** file. If the value of **TYPE(**l*dPointer***)** is the archive magic number**ARTYPE**, the flags field can be checked for the archive type. Large format archives will have the flag bit **AR_TYPE_BIG** set in **LDFLAGS(***ldPointer***)**.

Parameters

Item	Description
FileName	Specifies the file name of an object file or archive.
ldPointer	Points to an **LDFILE** structure.
FileDescriptor	Specifies a valid open file descriptor.
type	Points to a character string specifying the mode for the open file. The **fdopen** function is used to open the file.

Error Codes

Both the **ldopen** and **ldaopen** subroutines open the file named by the *FileName* parameter for reading. Both functions return a null value if the *FileName* parameter cannot be opened, or if memory for the **LDFILE** structure cannot be allocated.

A successful open does not ensure that the given file is a common object file or an archived object file.

Examples

The following is an example of code that uses the **ldopen** and **ldclose** subroutines:

```
/* for each FileName to be processed */

 ldPointer = NULL;
 do

 if((ldPointer = ldopen(FileName, ldPointer)) != NULL)

                      /* check magic number */
                      /* process the file */
                 "
                 "
   while(ldclose(ldPointer) == FAILURE );
```

ldrseek or ldnrseek Subroutine

Purpose

Seeks to the relocation entries of a section of an XCOFF file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldrseek ( ldPointer,  SectionIndex)
ldfile *ldPointer;
unsigned short SectionIndex;

int ldnrseek (ldPointer,  SectionName)
ldfile *ldPointer;
char *SectionName;
```

Description

The **ldrseek** subroutine seeks to the relocation entries of the section specified by the *SectionIndex* parameter of the common object file currently associated with the *ldPointer* parameter.

The **ldnrseek** subroutine seeks to the relocation entries of the section specified by the *SectionName* parameter.

The **ldrseek** subroutine and the **ldnrseek** subroutine determine the object mode of the associated file before seeking to the relocation entries of the indicated section.

Parameters

Item	Description
ldPointer	Points to an **LDFILE** structure that was returned as the result of a successful call to the **ldopen**, **lddopen**, or **ldaopen** subroutines.
SectionIndex	Specifies an index for the section whose relocation entries are to be sought.
SectionName	Specifies the name of the section whose relocation entries are to be sought.

Return Values

The **ldrseek** and **ldnrseek** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldrseek** subroutine fails if the contents of the *SectionIndex* parameter are greater than the number of sections in the object file. The **ldnrseek** subroutine fails if there is no section name corresponding with the *SectionName* parameter. Either function fails if the specified section has no relocation entries or if it cannot seek to the specified relocation entries.

Note: The first section has an index of 1.

ldshread or ldnshread Subroutine

Purpose

Reads a section header of an XCOFF file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldshread (ldPointer,SectionIndex,SectionHead)
LDFILE *ldPointer;
unsigned short SectionIndex;
void *SectionHead;

int ldnshread (ldPointer,SectionName, SectionHead)
LDFILE *ldPointer;
char *SectionName;
void *SectionHead;
```

Description

The **ldshread** subroutine reads the section header specified by the *SectionIndex* parameter of the common object file currently associated with the *ldPointer* parameter into the area of memory beginning at the location specified by the *SectionHead* parameter.

The **ldnshread** subroutine reads the section header named by the *SectionName* argument into the area of memory beginning at the location specified by the *SectionHead* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the section header of the associated object file. Since the **ldopen** subroutine provides magic number information (via the **HEADER(***ldPointer* **).f_magic** macro), the calling application can always determine whether the *SectionHead* pointer should refer to a 32-bit **SCNHDR** or 64-bit **SCNHDR_64** structure.

Only the first section header named by the *SectionName* argument is returned by the **ldshread** subroutine.

Parameters

Item	Description
ldPointer	Points to an **LDFILE** structure that was returned as the result of a successful call to the **ldopen**, **lldopen**, or **ldaopen** subroutine.
SectionIndex	Specifies the index of the section header to be read. **Note:** The first section has an index of 1.
SectionHead	Points to a buffer large enough to accept either a 32-bit or a 64-bit **SCNHDR** structure, according to the object mode of the file being read.
SectionName	Specifies the name of the section header to be read.

Return Values

The **ldshread** and **ldnshread** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldshread** subroutine fails if the *SectionIndex* parameter is greater than the number of sections in the object file. The **ldnshread** subroutine fails if there is no section with the name specified by the *SectionName* parameter. Either function fails if it cannot read the specified section header.

Examples

The following is an example of code that opens an object file, determines its mode, and uses the **ldnshread** subroutine to acquire the .text section header. This code would be compiled with both **__XCOFF32__** and **__XCOFF64__** defined:

```
#define __XCOFF32__
#define __XCOFF64__

#include <ldfcn.h>



/* for each FileName to be processed */

if ( (ldPointer = ldopen(FileName, ldPointer)) != NULL )
{
    SCNHDR    SectionHead32;
    SCNHDR_64 SectionHead64;
    void      *SectionHeader;

    if ( HEADER(ldPointer).f_magic == U802TOCMAGIC )
```

```
        SectionHeader = &SectionHead32;
    else if ( HEADER(ldPointer).f_magic == U803XTOCMAGIC )
        SectionHeader = &SectionHead64;
    else
        SectionHeader = NULL;

    if ( SectionHeader && (ldnshread( ldPointer, ".text", SectionHeader ) ==
SUCCESS) )
    {
        /* ...successfully read header... */
        /* ...process according to magic number... */
    }
}
```

ldsseek or ldnsseek Subroutine

Purpose

Seeks to an indexed or named section of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldsseek ( ldPointer,  SectionIndex)
LDFILE *ldPointer;
unsigned short SectionIndex;

int ldnsseek (ldPointer,  SectionName)
LDFILE *ldPointer;
char *SectionName;
```

Description

The **ldsseek** subroutine seeks to the section specified by the *SectionIndex* parameter of the common object file currently associated with the *ldPointer* parameter. The subroutine determines the object mode of the associated file before seeking to the indicated section.

The **ldnsseek** subroutine seeks to the section specified by the *SectionName* parameter.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.
SectionIndex	Specifies the index of the section whose line number entries are to be seeked to.
SectionName	Specifies the name of the section whose line number entries are to be seeked to.

Return Values

The **ldsseek** and **ldnsseek** subroutines return a SUCCESS or FAILURE value.

Error Codes

The **ldsseek** subroutine fails if the *SectionIndex* parameter is greater than the number of sections in the object file. The **ldnsseek** subroutine fails if there is no section name corresponding with the *SectionName* parameter. Either function fails if there is no section data for the specified section or if it cannot seek to the specified section.

Note: The first section has an index of 1.

ldtbindex Subroutine

Purpose

Computes the index of a symbol table entry of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

long ldtbindex ( ldPointer)
LDFILE *ldPointer;
```

Description

The **ldtbindex** subroutine returns the index of the symbol table entry at the current position of the common object file associated with the *ldPointer* parameter.

The index returned by the **ldtbindex** subroutine may be used in subsequent calls to the **ldtbread** subroutine. However, since the **ldtbindex** subroutine returns the index of the symbol table entry that begins at the current position of the object file, if the **ldtbindex** subroutine is called immediately after a particular symbol table entry has been read, it returns the index of the next entry.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as a result of a successful call to the **ldopen** or **ldaopen** subroutine.

Return Values

The **ldtbindex** subroutine returns the value BADINDEX upon failure. Otherwise a value greater than or equal to zero is returned.

Error Codes

The **ldtbindex** subroutine fails if there are no symbols in the object file or if the object file is not positioned at the beginning of a symbol table entry.

Note: The first symbol in the symbol table has an index of 0.

ldtbread Subroutine

Purpose

Reads an indexed symbol table entry of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldtbread ( ldPointer,  SymbolIndex,  Symbol)
LDFILE *ldPointer;
long SymbolIndex;
void *Symbol;
```

Description

The **ldtbread** subroutine reads the symbol table entry specified by the *SymbolIndex* parameter of the common object file currently associated with the *ldPointer* parameter into the area of memory beginning at the *Symbol* parameter. It is the responsibility of the calling routine to provide a pointer to a buffer large enough to contain the symbol table entry of the associated object file. Since the ldopen subroutine provides magic number information (via the **HEADER(***ldPointer***).f_magic** macro), the calling application can always determine whether the *Symbol* pointer should refer to a 32-bit **SYMENT** or 64-bit **SYMENT_64** structure.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.
SymbolIndex	Specifies the index of the symbol table entry to be read.
Symbol	Points to a either a 32-bit or a 64-bit **SYMENT** structure.

Return Values

The **ldtbread** subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **ldtbread** subroutine fails if the *SymbolIndex* parameter is greater than or equal to the number of symbols in the object file, or if it cannot read the specified symbol table entry.

Note: The first symbol in the symbol table has an index of 0.

ldtbseek Subroutine

Purpose

Seeks to the symbol table of a common object file.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
#include <stdio.h>
#include <ldfcn.h>

int ldtbseek ( ldPointer)
LDFILE *ldPointer;
```

Description

The **ldtbseek** subroutine seeks to the symbol table of the common object file currently associated with the *ldPointer* parameter.

Parameters

Item	Description
ldPointer	Points to the **LDFILE** structure that was returned as the result of a successful call to the **ldopen** or **ldaopen** subroutine.

Return Values

The **ldtbseek** subroutine returns a SUCCESS or FAILURE value.

Error Codes

The **ldtbseek** subroutine fails if the symbol table has been stripped from the object file or if the subroutine cannot seek to the symbol table.

leaveok Subroutine

Purpose

Controls physical cursor placement after a call to the **refresh** subroutine.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

leaveok( Window,  Flag)
WINDOW *Window;
bool Flag;
```

Description

The **leaveok** subroutine controls cursor placement after a call to the **refresh** (“refresh or wrefresh Subroutine” on page 1728) subroutine. If the *Flag* parameter is set to FALSE, curses leaves the physical cursor in the same location as logical cursor when the window is refreshed.

If the *Flag* parameter is set to TRUE, curses leaves the cursor as is and does not move the physical cursor when the window is refreshed. This option is useful for applications that do not use the cursor, because it reduces physical cursor motions.

By default **leaveok** is FALSE, and the physical cursor is moved to the same position as the logical cursor after a refresh.

Parameters

Item	Description
Flag	Specifies whether to leave the physical cursor alone after a refresh (TRUE) or to move the physical cursor to the logical cursor after a refresh (FALSE).
Window	Identifies the window to set the *Flag* parameter for.

Return Values

Item	Description
OK	Indicates the subroutine completed. The **leaveok** subroutine always returns this value.

Examples

1. To move the physical cursor to the same location as the logical cursor after refreshing the user-defined window `my_window`, enter:

```
WINDOW *my_window;
leaveok(my_window, FALSE);
```

2. To leave the physical cursor alone after refreshing the user-defined window `my_window`, enter:

```
WINDOW *my_window;
leaveok(my_window, TRUE);
```

lgamma, lgammaf, lgammal, lgammad32, lgammad64, and lgammad128 Subroutine

Purpose

Computes the log gamma.

Syntax

```
#include <math.h>

extern int signgam;

double lgamma (x)
double x;

float lgammaf (x)
float x;

long double lgammal (x)
long double x;
_Decimal32 lgammad32 (x)
_Decimal32 x;

_Decimal64 lgammad64 (x)
_Decimal64 x;
```

```
_Decimal128 lgammad128 (x)
_Decimal128 x;
```

Description

The sign of Gamma (*x*) is returned in the external integer **signgam** for the **lgamma**, **lgammaf**, and **lgammal** subroutines.

The **lgamma**, **lgammaf**, and **lgammal** subroutines are not reentrant. A function that is not required to be reentrant is not required to be thread-safe.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **lgamma**, **lgammaf**, **lgammal**, **lgammad32**, **lgammad64**, and **lgammad128** subroutines return the logarithmic gamma of *x*.

If *x* is a non-positive integer, a pole error shall occur and the **lgamma**, **lgammaf**, **lgammal**, **lgammad32**, **lgammad64**, and **lgammad128** subroutines will return +**HUGE_VAL**, +**HUGE_VALF**, +**HUGE_VALL**, +**HUGE_VAL_D32**, +**HUGE_VAL_D64**, and +**HUGE_VAL_D128** respectively.

If the correct value would cause overflow, a range error shall occur and the **lgamma**, **lgammaf**, **lgammal**, **lgammad32**, **lgammad64**, and **lgammad128** subroutines will return ±**HUGE_VAL**, ±**HUGE_VALF**, ±**HUGE_VALL**, +**HUGE_VAL_D32**, +**HUGE_VAL_D64**, and +**HUGE_VAL_D128** respectively.

If *x* is NaN, a NaN is returned.

If *x* is 1 or 2, +0 is returned.

If *x* is ±Inf, +Inf is returned.

lineout Subroutine

Purpose

Formats a print line.

Library

None (provided by the print formatter)

Syntax

```
#include <piostruct.h>

int lineout ( fileptr)
FILE *fileptr;
```

Description

The **lineout** subroutine is invoked by the formatter driver only if the **setup** subroutine returns a non-null pointer. This subroutine is invoked for each line of the document being formatted. The **lineout** subroutine reads the input data stream from the *fileptr* parameter. It then formats and outputs the print line until it recognizes a situation that causes vertical movement on the page.

The **lineout** subroutine should process all characters to be printed and all printer commands related to horizontal movement on the page.

The **lineout** subroutine should not output any printer commands that cause vertical movement on the page. Instead, it should update the **vpos** (new vertical position) variable pointed to by the **shars_vars** structure that it shares with the formatter driver to indicate the new vertical position on the page. It should also refresh the **shar_vars** variables for vertical increment and vertical decrement (reverse line-feed) commands.

When the **lineout** subroutine returns, the formatter driver sends the necessary commands to the printer to advance to the new vertical position on the page. This position is specified by the **vpos** variable. The formatter driver automatically handles top and bottom margins, new pages, initial pages to be skipped, and progress reports to the **qdaemon** daemon.

The following conditions can cause vertical movements:

- Line-feed control character or variable line-feed control sequence
- Vertical-tab control character
- Form-feed control character
- Reverse line-feed control character
- A line too long for the printer that wraps to the next line

Other conditions unique to a specific printer also cause vertical movement.

Parameters

Item	Description
fileptr	Specifies a file structure for the input data stream.

Return Values

Upon successful completion, the **lineout** subroutine returns the number of bytes processed from the input data stream. It excludes the end-of-file character and any control characters or escape sequences that result only in vertical movement on the page (for example, line feed or vertical tab).

If a value of 0 is returned and the value in the **vpos** variable pointed to by the **shars_vars** structure has not changed, or there are no more data bytes in the input data stream, the formatter driver assumes that printing is complete.

If the **lineout** subroutine detects an error, it uses the **piomsgout** subroutine to issue an error message. It then invokes the **pioexit** subroutine with a value of PIOEXITBAD.

Note: If either the **piocmdout** or **piogetstr** subroutine detects an error, it automatically issues its own error messages and terminates the print job.

link and linkat Subroutine

Purpose

Creates an additional directory entry for an existing file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
int link ( Path1,  Path2)
const char *Path1, *Path2;

int linkat ( DirFileDescriptor1,Path1,DirFileDescriptor2,Path2,Flag)
int DirFileDescriptor1, DirFileDescriptor2;
const char *Path1, *Path2;
int Flag;
```

Description

The **link** and **linkat** subroutines create an additional hard link (directory entry) for an existing file. Both the old and the new links share equal access rights to the underlying object.

The **linkat** subroutine is equivalent to the link subroutine if the *Flag* parameter has the **AT_SYMLINK_FOLLOW** bit set and if both the *DirFileDescriptor1* and *DirFileDescriptor2* parameters are **AT_FDCWD** or both the *Path1* and *Path2* parameters are absolute path names. If *DirFileDescriptor1* is a valid file descriptor of an open directory and *Path1* is a relative path name, *Path1* is considered to be relative to the directory that is associated with the *DirFileDescriptor1* parameter instead of the current working directory. The same applies to the *DirFileDescriptor2* and *Path2* parameters.

If either directory in the **linkat** subroutine was opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted for that directory by using the current permissions of the directory. If either directory was opened with the **O_SEARCH** open flag, the subroutine does not perform the check for that directory.

If the Flag parameter of the **linkat** subroutine does not have the **AT_SYMLINK_FOLLOW** bit set and the *Path1* parameter specifies a symbolic link, the subroutine creates a link to the symbolic link, not its target.

Parameters

Item	Description
Path1	Points to the path name of an existing file. If *DirFileDescriptor1* is specified and *Path1* is a relative path name, then *Path1* is considered relative to the directory specified by *DirFileDescriptor1*.
Path2	Points to the path name of the directory entry to be created. If *DirFileDescriptor2* is specified and *Path2* is a relative path name, then *Path2* is considered relative to the directory specified by *DirFileDescriptor2*.
DirFileDescriptor1	Specifies the file descriptor of an open directory.
DirFileDescriptor2	Specifies the file descriptor of an open directory.
Flag	Specifies a bit field. If it contains the **AT_SYMLINK_FOLLOW** bit and *Path1* points to a symbolic link, then the link is created to the file the symbolic link points at, else the link is created to the symbolic link.

Note:

1. If Network File System (NFS) is installed on your system, these paths can cross into another node.
2. With hard links, both the *Path1* and *Path2* parameters must reside on the same file system. Creating links to directories requires root user authority.

Return Values

Upon successful completion, the **link** and **linkat** subroutines return a value of 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **link** and **linkat** subroutines are unsuccessful if one of the following is true:

Item	Description
EACCES	Indicates the requested link requires writing in a directory that denies write permission.
EDQUOT	Indicates the directory in which the entry for the new link is being placed cannot be extended, or disk blocks could not be allocated for the link because the user or group quota of disk blocks or i-nodes on the file system containing the directory has been exhausted.
EEXIST	Indicates the link named by the *Path2* parameter already exists.
EMLINK	Indicates the file already has the maximum number of links.
ENOENT	Indicates the file named by the *Path1* parameter does not exist.
ENOSPC	Indicates the directory in which the entry for the new link is being placed cannot be extended because there is no space left on the file system containing the directory.
EPERM	Indicates the file named by the *Path1* parameter is a directory, and the calling process does not have root user authority.
EROFS	Indicates the requested link requires writing in a directory on a read-only file system.
EXDEV	Indicates the link named by the *Path2* parameter and the file named by the *Path1* parameter are on different file systems, or the file named by *Path1* refers to a named STREAM.

The **linkat** subroutine is unsuccessful if one or more of the following is true:

Item	Description
EBADF	The *Path1* or *Path2* parameter does not specify an absolute path and the corresponding *DirFileDescriptor1* or *DirFileDescriptor2* parameter is neither **AT_FDCWD** nor a valid file descriptor.
ENOTDIR	The *Path1* or *Path2* parameter does not specify an absolute path and the corresponding *DirFileDescriptor1* or *DirFileDescriptor2* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.
EINVAL	The value of the *Flag* parameter is not valid.

The **link** and **linkat** subroutines can be unsuccessful for other reasons.

If NFS is installed on the system, the **link** and **linkat** subroutines are unsuccessful if the following is true:

Item	Description
ETIMEDOUT	Indicates the connection timed out.

lio_listio or lio_listio64 Subroutine

The **lio_listio** or **lio_listio64** subroutine includes information for the POSIX AIO **lio_listio** subroutine (as defined in the IEEE std 1003.1-2001), and the Legacy AIO **lio_listio** subroutine.

POSIX AIO lio_listio Subroutine

Purpose

Initiates a list of asynchronous I/O requests with a single call.

Syntax

```
#include <aio.h>
int lio_listio(mode, list, nent, sig)
int mode;
struct aiocb *restrict const list[restrict];
int nent;
struct sigevent *restrict sig;
```

Description

The *lio_listio* subroutine initiates a list of I/O requests with a single function call.

The *mode* parameter takes one of the values (LIO_WAIT, LIO_NOWAIT or LIO_NOWAIT_AIOWAIT) declared in **<aio.h>** and determines whether the subroutine returns when the I/O operations have been completed, or as soon as the operations have been queued. If the *mode* parameter is set to LIO_WAIT, the subroutine waits until all I/O is complete and the *sig* parameter is ignored.

If the *mode* parameter is set to LIO_NOWAIT or LIO_NOWAIT_AIOWAIT, the subroutine returns immediately. If LIO_NOWAIT is set, asynchronous notification occurs, according to the *sig* parameter, when all I/O operations complete. If *sig* is NULL, no asynchronous notification occurs. If *sig* is not NULL, asynchronous notification occurs when all the requests in *list* have completed. If LIO_NOWAIT_AIOWAIT is set, the **aio_nwait** subroutine must be called for the aio control blocks to be updated.

The I/O requests enumerated by *list* are submitted in an unspecified order.

The *list* parameter is an array of pointers to **aiocb** structures. The array contains *nent* elements. The array may contain NULL elements, which are ignored.

The *aio_lio_opcode* field of each **aiocb** structure specifies the operation to be performed. The supported operations are LIO_READ, LIO_WRITE, and LIO_NOP; these symbols are defined in **<aio.h>**. The LIO_NOP operation causes the list entry to be ignored. If the *aio_lio_opcode* element is equal to LIO_READ, an I/O operation is submitted as if by a call to **aio_read** with the *aiocbp* equal to the address of the **aiocb** structure. If the *aio_lio_opcode* element is equal to LIO_WRITE, an I/O operation is submitted as if by a call to **aio_write** with the *aiocbp* argument equal to the address of the **aiocb** structure.

The *aio_fildes* member specifies the file descriptor on which the operation is to be performed.

The *aio_buf* member specifies the address of the buffer to or from which the data is transferred.

The *aio_nbytes* member specifies the number of bytes of data to be transferred.

The members of the **aiocb** structure further describe the I/O operation to be performed, in a manner identical to that of the corresponding **aiocb** structure when used by the **aio_read** and **aio_write** subroutines.

The *nent* parameter specifies how many elements are members of the list.

The behavior of the **lio_listio** subroutine is altered according to the definitions of synchronized I/O data integrity completion and synchronized I/O file integrity completion if synchronized I/O is enabled on the file associated with *aio_fildes* .

For regular files, no data transfer occurs past the offset maximum established in the open file description.

Parameters

mode
Determines whether the subroutine returns when the I/O operations are completed, or as soon as the operations are queued.

list
An array of pointers to aio control structures defined in the **aio.h** file.

nent
Specifies the length of the array.

sig
Determines when asynchronous notification occurs.

Execution Environment

The **lio_listio** and **lio_listio64** subroutines can be called from the **process environment** only.

Return Values

When the **lio_listio** subroutine is successful, it returns a value of 0. Otherwise, it returns a value of -1 and sets the **errno** global variable to identify the error. The returned value indicates the success or failure of the **lio_listio** subroutine itself and not of the asynchronous I/O requests (except when the command is **LIO_WAIT**). The **aio_error** subroutine returns the status of each I/O request. The possible **errno** values are as follows:

EAGAIN
The resources necessary to queue all the I/O requests were not available. The application may check the error status of each **aiocb** to determine the individual request(s) that failed.

The number of entries indicated by *nent* would cause the system-wide limit (AIO_MAX) to be exceeded.

EINVAL
The *mode* parameter is not a proper value, or the value of *nent* was greater than AIO_LISTIO_MAX.

EINTR
A signal was delivered while waiting for all I/O requests to complete during an LIO_WAIT operation. Since each I/O operation invoked by the **lio_listio** subroutine may provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited. Outstanding I/O requests are not canceled, and the application examines each list element to determine whether the request was initiated, canceled, or completed.

EIO
One or more of the individual I/O operations failed. The application may check the error status for each **aiocb** structure to determine the individual request(s) that failed.

If the **lio_listio** subroutine succeeds or fails with errors of **EAGAIN**, **EINTR**, or **EIO**, some of the I/O specified by the list may have been initiated. If the **lio_listio** subroutine fails with an error code other than **EAGAIN**, **EINTR**, or **EIO**, no operations from the list were initiated. The I/O operation indicated by each list element can encounter errors specific to the individual read or write function being performed. In this event, the error status for each *aiocb* control block contains the associated error code. The error codes that can be set are the same as would be set by the **read** or **write** subroutines, with the following additional error codes possible:

EAGAIN
The requested I/O operation was not queued due to resource limitations.

ECANCELED
The requested I/O was canceled before the I/O completed due to an **aio_cancel** request.

EFBIG
The *aio_lio_opcode* argument is LIO_WRITE, the file is a regular file, *aio_nbytes* is greater than 0, and *aio_offset* is greater than or equal to the offset maximum in the open file description associated with *aio_fildes*.

EINPROGRESS
The requested I/O is in progress.

EOVERFLOW
The *aio_lio_opcode* argument is set to LIO_READ, the file is a regular file, *aio_nbytes* is greater than 0, and the *aio_offset* argument is before the end-of-file and is greater than or equal to the offset maximum in the open file description associated with *aio_fildes*.

Legacy AIO lio_listio Subroutine

Purpose

Initiates a list of asynchronous I/O requests with a single call.

Syntax

```
#include <aio.h>
```

```
int lio_listio (cmd,
list, nent, eventp)
int  cmd,  nent;
struct liocb * list[ ];
struct event * eventp;
```

```
int lio_listio64
(cmd, list,nent, eventp)
int cmd, nent; struct liocb64 *list;
struct event *eventp;
```

Description

The **lio_listio** subroutine allows the calling process to initiate the *nent* parameter asynchronous I/O requests. These requests are specified in the **liocb** structures pointed to by the elements of the *list* array. The call may block or return immediately depending on the *cmd* parameter. If the *cmd* parameter requests that I/O completion be asynchronously notified, a **SIGIO** signal is delivered when all I/O operations are completed.

The **lio_listio64** subroutine is similar to the **lio_listio** subroutine except that it takes an array of pointers to **liocb64** structures. This allows the **lio_listio64** subroutine to specify offsets in excess of OFF_MAX (2 gigbytes minus 1).

In the large file enabled programming environment, **lio_listio** is redefined to be **lio_listio64**.

Note: The pointer to the **event** structure *eventp* parameter is currently not in use, but is included for future compatibility.

Parameters

cmd

The *cmd* parameter takes one of the following values:

LIO_WAIT
Queues the requests and waits until they are complete before returning.

LIO_NOWAIT
Queues the requests and returns immediately, without waiting for them to complete. The *event* parameter is ignored.

LIO_NOWAIT_AIOWAIT
Queues the requests and returns immediately, without waiting for them to complete. The **aio_nwait** subroutine must be called for the aio control blocks to be updated. Use of the **aio_suspend** subroutine and the **aio_cancel** subroutine on these requests are not supported, nor is any form of asynchronous notification for individual requests.

LIO_ASYNC
Queues the requests and returns immediately, without waiting for them to complete. An enhanced signal is delivered when all the operations are completed. Currently this command is not implemented.

LIO_ASIG
Queues the requests and returns immediately, without waiting for them to complete. A **SIGIO** signal is generated when all the I/O operations are completed.

LIO_NOWAIT_GMCS
Queues the requests and returns immediately, without waiting for them to complete. The **GetMultipleCompletionStatus** subroutine must be called to retrieve the completion status for

the requests. The aio control blocks are not updated. Use of the **aio_suspend** subroutine and the **aio_cancel** subroutine on these requests are not supported, nor is any form of asynchronous notification.

list
Points to an array of pointers to **liocb** structures. The structure array contains *nent* elements:

lio_aiocb
The asynchronous I/O control block associated with this I/O request. This is an actual **aiocb** structure, not a pointer to one.

lio_fildes
Identifies the file object on which the I/O is to be performed.

lio_opcode
This field may have one of the following values defined in the **/usr/include/sys/aio.h** file:

LIO_READ
Indicates that the read I/O operation is requested.

LIO_WRITE
Indicates that the write I/O operation is requested.

LIO_NOP
Specifies that no I/O is requested (that is, this element will be ignored).

nent
Specifies the length of the array.

eventp
Points to an **event** structure to be used when the *cmd* parameter is set to the **LIO_ASYNC** value. This parameter is currently ignored.

Execution Environment

The **lio_listio** and **lio_listio64** subroutines can be called from the **process environment** only.

Return Values

When the **lio_listio** subroutine is successful, it returns a value of 0. Otherwise, it returns a value of -1 and sets the **errno** global variable to identify the error. The returned value indicates the success or failure of the **lio_listio** subroutine itself and not of the asynchronous I/O requests (except when the command is **LIO_WAIT**). The **aio_error** subroutine returns the status of each I/O request.

If the `lio_listio` subroutine succeeds or fails with errors of `EAGAIN`, `EINTR`, or `EIO`, some of the I/O specified by the list might have been initiated. If the `lio_listio` subroutine fails with an error code other than `EAGAIN`, `EINTR`, or `EIO`, no operations from the list were initiated. The I/O operation indicated by each list element can encounter errors specific to the individual read or write function being performed. In this event, the error status for each `aiocb` control block contains the associated error code. The error codes that can be set are the same as would be set by the read or write subroutines, with the following additional error codes possible:

EAGAIN
Indicates that the system resources required to queue the request are not available. Specifically, the transmit queue may be full, or the maximum number of opens may have been reached.

EINTR
Indicates that a signal or event interrupted the **lio_listio** subroutine call.

EINVAL
Indicates that the `aio_whence` field does not have a valid value or that the resulting pointer is not valid.

EIO
One or more of the individual I/O operations failed. The application can check the error status for each `aiocb` structure to determine the individual request that failed.

listea Subroutine

Purpose

Lists the extended attributes associated with a file.

Syntax

```
#include <sys/ea.h>

ssize_t listea(const char *path, char *list, size_t size);
ssize_t flistea (int filedes, char *list, size_t size);
ssize_t llistea (const char *path, char *list, size_t size);
```

Description

Extended attributes are name:value pairs associated with the file system objects (such as files, directories, and symlinks). They are extensions to the normal attributes that are associated with all objects in the file system (that is, the **stat(2)** data).

Do not define an extended attribute name with eight characters prefix "(0xF8)SYSTEM(0xF8)". Prefix "(0xF8)SYSTEM(0xF8)" is reserved for system use only.

Note: The 0xF8 prefix represents a non-printable character.

The **listea** subroutine retrieves the list of extended attribute names associated with the given *path* in the file system. The *list* is the set of (NULL-terminated) names, one after the other. Names of extended attributes to which the calling process does not have access might be omitted from the list. The length of the attribute name list is returned. The **flistea** subroutine is identical to **listea**, except that it takes a file descriptor instead of a path. The **llistea** subroutine is identical to **listea**, except, in the case of a symbolic link, the link itself is interrogated, not the file that it refers to.

An empty buffer of size 0 can be passed into these calls to return the current size of the list of extended attribute names, which can be used to estimate whether the size of a buffer is sufficiently large to hold the list of names.

Parameters

Item	Description
path	The path name of the file.
list	A pointer to a buffer in which the list of attributes will be stored.
size	The size of the buffer.
filedes	A file descriptor for the file.

Return Values

If the **listea** subroutine succeeds, a nonnegative number is returned that indicates the length in bytes of the attribute name list. Upon failure, -1 is returned and **errno** is set appropriately.

Error Codes

Item	Description
EACCES	Caller lacks read permission on the base file, or lacks the appropriate ACL privileges for named attribute **read**.
EFAULT	A bad address was passed for *path* or *list*.

Item	Description
EFORMAT	File system is capable of supporting EAs, but EAs are disabled.
ENOTSUP	Extended attributes are not supported by the file system.
ERANGE	The size of the list buffer is too small to hold the result.

llrint, llrintf, llrintl, llrintd32, llrintd64, and llrintd128 Subroutines

Purpose

Round to the nearest integer value using current rounding direction.

Syntax

```
#include <math.h>

long long llrint (x)
double x;

long long llrintf (x)
float x;

long long llrintl (x)
long double x;

long long llrintd32(x)
_Decimal32 x;

long long llrintd64(x)
_Decimal64 x;

long long llrintd128(x)
_Decimal128 x;
```

Description

The **llrint**, **llrintf**, **llrintl**, **llrintd32**, **llrintd64**, and **llrintd128** subroutines round the *x* parameter to the nearest integer value, according to the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be rounded.

Return Values

Upon successful completion, the **llrint**, **llrintf**, **llrintl**, **llrintd32**, **llrintd64**, and **llrintd128** subroutines return the rounded integer value.

If *x* is NaN, a domain error occurs, and an unspecified value is returned.

If *x* is +Inf, a domain error occurs and an unspecified value is returned.

If *x* is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long long**, a domain error occur and an unspecified value is returned.

If the correct value is negative and too large to represent as a **long long**, a domain error occurs and an unspecified value is returned.

llround, llroundf, llroundl, llroundd32, llroundd64, and llroundd128 Subroutines

Purpose

Round to the nearest integer value.

Syntax

```
#include <math.h>

long long llround (x)
double x;

long long llroundf (x)
float x;

long long llroundl (x)
long double x;

long long llroundd32(x)
_Decimal32 x;

long long llroundd64(x)
_Decimal64 x;

long long llroundd128(x)
_Decimal128 x;
```

Description

The **llround**, **llroundf**, **llroundl**, **llroundd32**, **llroundd64**, and **llroundd128** subroutines round the *x* parameter to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be rounded.

Return Values

Upon successful completion, the **llround**, **llroundf**, **llroundl**, **llroundd32**, **llroundd64**, and **llroundd128** subroutines return the rounded integer value.

If *x* is NaN, a domain error occurs, and an unspecified value is returned.

If *x* is +Inf, a domain error occurs and an unspecified value is returned.

If *x* is –Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long long**, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a **long long**, a domain error occurs and an unspecified value is returned.

load and loadAndInit Subroutines

Purpose

Loads a module into the current process.

Syntax

```
int *load ( ModuleName, Flags, LibraryPath)
char *ModuleName;
uint Flags;
char *LibraryPath;
```

```
int *loadAndInit ( ModuleName, Flags, LibraryPath)
char *ModuleName;
uint Flags;
char *LibraryPath;
```

Description

The **load** and `loadAndInit` subroutines load the specified module into the calling process's address space. A module can be a regular file or a member of an archive. When adding a new module to the address space of a 32-bit process, the load operation may cause the break value to change.

The **load** subroutine is not a preferred method to load C++ modules. Use **loadAndInit** subroutine instead. The **loadAndInit** subroutine uses the same interface as **load** but performs C++ initialization.

The **exec** subroutine is similar to the **load** subroutine, except that:

- The **load** subroutine does not replace the current program with a new one.
- The **exec** subroutine does not have an explicit library path parameter; it has only the **LIBPATH** and `LD_LIBRARY_PATH` environment variables. Also, these library path environment variables are ignored when the program using the **exec** subroutine has more privilege than the caller (for example, in the case of a **set-UID** program).

A large application can be split up into one or more modules in one of two ways that allow execution within the same process. The first way is to create each of the application's modules separately and use **load** to explicitly load a module when it is needed. The other way is to specify the relationship between the modules when they are created by defining imported and exported symbols.

Modules can import symbols from other modules. Whenever symbols are imported from one or more other modules, these modules are automatically loaded to resolve the symbol references if the required modules are not already loaded, and if the imported symbols are not specified as deferred imports. These modules can be archive members in libraries or individual files and can have either shared or private file characteristics that control how and where they are loaded.

Shared modules (typically members of a shared library archive) are loaded into the shared library region, when their access permissions allow sharing, that is, when they have read-other permission. Private modules, and shared modules without the required permissions for sharing, are loaded into the process private region.

When the loader resolves a symbol, it uses the file name recorded with that symbol to find the module that exports the symbol. If the file name contains any **/** (slash) characters, it is used directly and must name an appropriate file or archive member. However, if the file name is a base name (contains no **/** characters), the loader searches the directories specified in the default library path for a file (i.e. a module or an archive) with that base name.

The *LibraryPath* is a string containing one or more directory path names separated by colons. See the section “Searching for Dependent Modules” on page 861 for information on library path searching.

When a process is executing under **ptrace** control, portions of the process's address space are recopied after the **load** processing completes. For a 32-bit process, the main program text (loaded in segment 1) and shared library modules (loaded in segment 13) are recopied. Any breakpoints or other modifications to these segments must be reinserted after the **load** call. For a 64-bit process, shared library modules are recopied after a **load** call. The debugger will be notified by setting the **W_SLWTED** flag in the status returned by **wait**, so that it can reinsert breakpoints.

When a process executing under **ptrace** control calls **load**, the debugger is notified by setting the **W_SLWTED** flag in the status returned by **wait**. Any modules newly loaded into the shared library segments will be copied to the process's private copy of these segments, so that they can be examined or modified by the debugger.

The **load** subroutine will call initialization routines (**init** routines) for the new module and any of its dependents if they were not already loaded.

Modules loaded by this subroutine are automatically unloaded when the process terminates or when the **exec** subroutine is executed. They are explicitly unloaded by calling the **unload** subroutine.

Searching for Dependent Modules

The load operation and the exec operation differ slightly in their dependent module search mechanism. When a module is added to the address space of a running process (the load operation), the rules outlined in the next section are used to find the named module. Note that dependency relationships may be loosely defined as a tree but recursive relationships between modules may also exist. The following components may used to create a complete library search path:

1. If the **L_LIBPATH_EXEC** flag is set, the library search path used at exec-time.
2. The value of the *LibraryPath* parameter if it is non-null. Note that a null string is a valid search path which refers to the current working directory. If the *LibraryPath* parameter is NULL, the value of the **LIBPATH** environment variable, or alternatively the LD_LIBRARY_PATH environment variable (if LIBPATH is not set), is used instead.
3. The library search path contained in the loader section of the module being loaded (the *ModuleName* parameter).
4. The library search path contained in the loader section of the module whose immediate dependents are being loaded. Note that this per-module information changes when searching for each module's immediate dependents.

To find the *ModuleName* module, components 1 and 2 are used. To find dependents, components 1, 2, 3 and 4 are used in order. Note that if any modules that are already part of the running process satisfy the dependency requirements of the newly loaded module(s), pre-existing modules are not loaded again.

For each colon-separated portion of the aggregate search specification, if the base name is not found the search continues. Additionally, if the needed file is not an archive member, the search will continue past a file having the wrong object mode. If an archive member is needed, searching stops when the first match of the file name is found. If the file is not of the proper form, or in the case of an archive that does not contain the required archive member, or does not export a definition of a required symbol, an error occurs. The library path search is not performed when either a relative or an absolute path name is specified for a dependent module.

The library search path stored within the module is specified at link-edit time.

The **load** subroutine may cause the calling process to fail if the module specified has a very long chain of dependencies, (for example, lib1.a, which depends on lib2.a, which depends on lib3.a, etc). This is because the loader processes such relationships recursively on a fixed-size stack. This limitation is exposed only when processing a dependency chain that has over one thousand elements.

Parameters

Item	Description
ModuleName	Points to the name of the module to be loaded. The module name consists of a path name, and, an optional member name. If the path name contains at least on / character, the name is used directly, and no directory searches are performed to locate the file. If the path name contains no / characters, it is treated as a base name, and should be in one of the directories listed in the library path. The library path is either the value of the *LibraryPath* parameter if not a null value, or the value of the **LIBPATH** environment variable (if set; otherwise, LD_LIBRARY_PATH environment variable, if set) or the library path used at process exec time (if the **L_LIBPATH_EXEC** is set). If no library path is provided, the module should be in the current directory. The *ModuleName* parameter may explicitly name an archive member. The syntax is *pathname*(*member*) where *pathname* follows the rules specified in the previous paragraph, and *member* is the name of a specific archive member. The parentheses are a required portion of the specification and no intervening spaces are allowed. If an archive member is named, the **L_LOADMEMBER** flag must be added to the *Flags* parameter. Otherwise, the entire *ModuleName* parameter is treated as an explicit filename.
Flags	Modifies the behavior of the **load** and the loadAndInit services as follows (see the **ldr.h** file). If no special behavior is required, set the value of the flags parameter to 0 (zero). For compatibility, a value of 1 (one) may also be specified. **L_LIBPATH_EXEC** Specifies that the library path used at process exec time should be prepended to any library path specified in the **load** call (either as an argument or environment variable). It is recommended that this flag be specified in all calls to the **load** subroutine. **L_LOADMEMBER** Indicates that the *ModuleName* parameter may specify an archive member. The *ModuleName* argument is searched for parentheses, and if found the parameter is treated as a filename/member name pair. If this flag is present and the *ModuleName* parameter does not contain parenthesis the entire *ModuleName* parameter is treated as a filename specification. Under either condition the filename is expected to be found within the library path or the current directory. **L_NOAUTODEFER** Specifies that any deferred imports in the module being loaded must be explicitly resolved by use of the **loadbind** subroutine. This allows unresolved imports to be explicitly resolved at a later time with a specified module. If this flag is not specified, deferred imports (marked for deferred resolution) are resolved at the earliest opportunity when any subsequently loaded module exports symbols matching unresolved imports.
LibraryPath	Points to a character string that specifies the default library search path. If the *LibraryPath* parameter is NULL, the **LIBPATH** environment variable is used, if set; otherwise, the LD_LIBRARY_PATH environment variable is used. The library path is used to locate dependent modules that are specified as basenames (that is, their pathname components do not contain a / (slash) character. Note the difference between setting the *LibraryPath* parameter to null, and having the *LibraryPath* parameter point to a null string (" "). A null string is a valid library path which consists of a single directory: the current directory.

Return Values

Upon successful completion, the **load** and `loadAndInit` subroutines return the pointer to function for the entry point of the module. If the module has no entry point, the address of the data section of the module is returned.

Error Codes

If the **load** and `loadAndInit` subroutines fail, a null pointer is returned, the module is not loaded, and **errno** global variable is set to indicate the error. The **load** and `loadAndInit` subroutines fail if one or more of the following are true of a module to be explicitly or automatically loaded:

Item	Description
EACCES	Indicates the file is not an ordinary file, or the mode of the program file denies execution permission, or search permission is denied on a component of the path prefix.
EINVAL	Indicates the file or archive member has a valid magic number in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.
ELOOP	Indicates too many symbolic links were encountered in translating the path name.
ENOEXEC	Indicates an error occurred when loading or resolving symbols for the specified module. This can be due to an attempt to load a module with an invalid **XCOFF** header, a failure to resolve symbols that were not defined as deferred imports or several other load time related problems. The **loadquery** subroutine can be used to return more information about the load failure. If runtime linking is used, the **load** and the **loadAndInit** subroutines will fail if the runtime linker could not resolve some symbols. In this case, **errno** will be set to **ENOEXEC**, but the **loadquery** subroutine will not return any additional information.
ENOMEM	Indicates the program requires more memory than is allowed by the system-imposed maximum.
ETXTBSY	Indicates the file is currently open for writing by some process.
ENAMETOOLONG	Indicates a component of a path name exceeded 255 characters, or an entire path name exceeded 1023 characters.
ENOENT	Indicates a component of the path prefix does not exist, or the path name is a null value. For the **dlopen** subroutine, RTLD_MEMBER is not used when trying to open a member within the archive file.
ENOTDIR	Indicates a component of the path prefix is not a directory.
ESTALE	Indicates the process root or current directory is located in a virtual file system that has been unmounted.

loadbind Subroutine

Purpose

Provides specific run-time resolution of a module's deferred symbols.

Syntax

```
int loadbind( Flag,  ExportPointer,  ImportPointer)
int Flag;
void *ExportPointer, *ImportPointer;
```

Description

The **loadbind** subroutine controls the run-time resolution of a previously loaded object module's unresolved imported symbols.

The **loadbind** subroutine is used when two modules are loaded. Module A, an object module loaded at run time with the **load** subroutine, has designated that some of its imported symbols be resolved at a later time. Module B contains exported symbols to resolve module A's unresolved imports.

To keep module A's imported symbols from being resolved until the **loadbind** service is called, you can specify the **load** subroutine flag, **L_NOAUTODEFER**, when loading module A.

When a 32-bit process is executing under **ptrace** control, portions of the process's address space are recopied after the **loadbind** processing completes. The main program text (loaded in segment 1) and shared library modules (loaded in segment 13) are recopied. Any breakpoints or other modifications to these segments must be reinserted after the **loadbind** call.

When a 32-bit process executing under **ptrace** control calls **loadbind**, the debugger is notified by setting the **W_SLWTED** flag in the status returned by **wait**.

When a 64-bit process under **ptrace** control calls **loadbind**, the debugger is not notified and execution of the process being debugged continues normally.

Parameters

Item	Description
Flag	Currently not used.
ExportPointer	Specifies the function pointer returned by the **load** subroutine when module B was loaded.
ImportPointer	Specifies the function pointer returned by the **load** subroutine when module A was loaded.

Note: The *ImportPointer* or *ExportPointer* parameter may also be set to any exported static data area symbol or function pointer contained in the associated module. This would typically be the function pointer returned from the **load** of the specified module.

Return Values

A 0 is returned if the **loadbind** subroutine is successful.

Error Codes

A -1 is returned if an error is detected, with the **errno** global variable set to an associated error code:

Item	Description
EINVAL	Indicates that either the *ImportPointer* or *ExportPointer* parameter is not valid (the pointer to the *ExportPointer* or *ImportPointer* parameter does not correspond to a loaded program module or library).
ENOMEM	Indicates that the program requires more memory than allowed by the system-imposed maximum.

After an error is returned by the **loadbind** subroutine, you may also use the **loadquery** subroutine to obtain additional information about the **loadbind** error.

loadquery Subroutine

Purpose

Returns error information from the **load** or **exec** subroutine; also provides a list of object files loaded for the current process.

Syntax

```
int loadquery( Flags,  Buffer,  BufferLength)
int Flags;
void *Buffer;
unsigned int BufferLength;
```

Description

The **loadquery** subroutine obtains detailed information about an error reported on the last **load** or **exec** subroutine executed by a calling process. The **loadquery** subroutine may also be used to obtain a list of object file names for all object files that have been loaded for the current process, or the library path that was used at process exec time.

Parameters

Item	Description
Buffer	Points to a *Buffer* in which to store the information.
BufferLength	Specifies the number of bytes available in the *Buffer* parameter.

Item	Description

Flags	Specifies the action of the **loadquery** subroutine as follows:
	L_GETINFO
	Returns a list of all object files loaded for the current process, and stores the list in the *Buffer* parameter. The object file information is contained in a sequence of **LD_INFO** structures as defined in the **sys/ldr.h** file. Each structure contains the module location in virtual memory and the path name that was used to load it into memory. The file descriptor field in the **LD_INFO** structure is not filled in by this function.
	L_GETMESSAGE
	Returns detailed error information describing the failure of a previously invoked **load** or **exec** function, and stores the error message information in *Buffer.* Upon successful return from this function the beginning of the *Buffer* contains an array of character pointers. Each character pointer points to a string in the buffer containing a loader error message. The character array ends with a null character pointer. Each error message string consists of an ASCII message number followed by zero or more characters of error-specific message data. Valid message numbers are listed in the **sys/ldr.h** file.
	You can format the error messages returned by the **L_GETMESSAGE** function and write them to standard error using the standard system command **/usr/sbin/execerror** as follows:
	`char *buffer[1024];`
	`buffer[0] = "execerror";`
	`buffer[1] = "name of program that failed to load";`
	`loadquery(L_GETMESSAGES, &buffer[2],\`
	`  sizeof buffer-2*sizeof(char*));`
	`execvp("/usr/sbin/execerror",buffer);`
	This sample code causes the application to terminate after the messages are written to standard error.
	L_GETLIBPATH
	Returns the library path that was used at process exec time. The library path is a null terminated character string.
	L_GETXINFO
	Returns a list of all object files loaded for the current process and stores the list in the *Buffer* parameter. The object file information is contained in a sequence of **LD_XINFO** structures as defined in the **sys/ldr.h** file. Each structure contains the module location in virtual memory and the path name that was used to load it into memory. The file descriptor field in the **LD_XINFO** structure is not filled in by this function.

Return Values

Upon successful completion, **loadquery** returns the requested information in the caller's buffer specified by the *Buffer* and *BufferLength* parameters.

Error Codes

The **loadquery** subroutine returns with a return code of -1 and the **errno** global variable is set to one of the following when an error condition is detected:

Item	Description
ENOMEM	Indicates that the caller's buffer specified by the *Buffer* and *BufferLength* parameters is too small to return the information requested. When this occurs, the information in the buffer is undefined.

Item	Description
EINVAL	Indicates the function specified in the *Flags* parameter is not valid.
EFAULT	Indicates the address specified in the *Buffer* parameter is not valid.

localeconv Subroutine

Purpose

Sets the locale-dependent conventions of an object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <locale.h>
```

```
struct lconv *localeconv ( )
```

Description

The **localeconv** subroutine sets the components of an object using the **lconv** structure. The **lconv** structure contains values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale.

The fields of the structure with the type **char *** are strings, any of which (except `decimal_point`) can point to a null string, which indicates that the value is not available in the current locale or is of zero length. The fields with type **char** are nonnegative numbers, any of which can be the **CHAR_MAX** value which indicates that the value is not available in the current locale. The fields of the **Iconv** structure include the following:

Item	Description
`char *decimal_point`	The decimal-point character used to format non-monetary quantities.
`char *thousands_sep`	The character used to separate groups of digits to the left of the decimal point in formatted non-monetary quantities.
`char *grouping`	A string whose elements indicate the size of each group of digits in formatted non-monetary quantities. The value of the `grouping` field is interpreted according to the following: **CHAR_MAX** No further grouping is to be performed. **0** The previous element is to be repeatedly used for the remainder of the digits. **other** The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

Item	Description
`char *int_curr_symbol`	The international currency symbol applicable to the current locale, left-justified within a four-character space-padded field. The character sequences are in accordance with those specified in ISO 4217, "Codes for the Representation of Currency and Funds."
`char *currency_symbol`	The local currency symbol applicable to the current locale.
`char *mon_decimal_point`	The decimal point used to format monetary quantities.
`char *mon_thousands_sep`	The separator for groups of digits to the left of the decimal point in formatted monetary quantities.
`char *mon_grouping`	A string whose elements indicate the size of each group of digits in formatted monetary quantities. The value of the `mon_grouping` field is interpreted according to the following: **CHAR_MAX** No further grouping is to be performed. **0** The previous element is to be repeatedly used for the remainder of the digits. **other** The value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.
`char *positive_sign`	The string used to indicate a nonnegative formatted monetary quantity.
`char *negative_sign`	The string used to indicate a negative formatted monetary quantity.
`char int_frac_digits`	The number of fractional digits (those to the right of the decimal point) to be displayed in a formatted monetary quantity.
`char p_cs_precedes`	Set to 1 if the specified currency symbol (the `currency_symbol` or `int_curr_symbol` field) precedes the value for a nonnegative formatted monetary quantity and set to 0 if the specified currency symbol follows the value for a nonnegative formatted monetary quantity.
`char p_sep_by_space`	Set to 1 if the `currency_symbol` or `int_curr_symbol` field is separated by a space from the value for a nonnegative formatted monetary quantity and set to 0 if the `currency_symbol` or `int_curr_symbol` field is not separated by a space from the value for a nonnegative formatted monetary quantity.
`char n_cs_precedes`	Set to 1 if the `currency_symbol` or `int_curr_symbol` field precedes the value for a negative formatted monetary quantity and set to 0 if the `currency_symbol` or `int_curr_symbol` field follows the value for a negative formatted monetary quantity.

Item	Description
`char n_sep_by_space`	Set to 1 if the `currency_symbol` or `int_curr_symbol` field is separated by a space from the value for a negative formatted monetary quantity and set to 0 if the `currency_symbol` or `int_curr_symbol` field is not separated by a space from the value for a negative formatted monetary quantity. Set to 2 if the symbol and the sign string are adjacent and separated by a blank character.
`char p_sign_posn`	Set to a value indicating the positioning of the positive sign (the `positive_sign` fields) for nonnegative formatted monetary quantity.
`char n_sign_posn`	Set to a value indicating the positioning of the negative sign (the `negative_sign` fields) for a negative formatted monetary quantity. The values of the `p_sign_posn` and `n_sign_posn` fields are interpreted according to the following definitions: **0** Parentheses surround the quantity and the specified currency symbol or international currency symbol. **1** The sign string precedes the quantity and the currency symbol or international currency symbol. **2** The sign string follows the quantity and currency symbol or international currency symbol. **3** The sign string immediately precedes the currency symbol or international currency symbol. **4** The sign string immediately follows the currency symbol or international currency symbol.

The following table illustrates the rules that can be used by three countries to format monetary quantities:

Country	Formats
Italy	**Positive Format:** L.1234 **Negative Format:** -L.1234 **International Format:** ITL.1234
Norway	**Positive Format:** krl.234.56 **Negative Format:** krl.234.56- **International Format:** NOK 1.234.56

Country	Formats
Switzerland	**Positive Format:** SFrs.1.234.56 **Negative Format:** SFrs.1.234.56C **International Format:** CHF 1.234.56

The following table shows the values of the monetary members of the structure returned by the **localeconv** subroutine for these countries:

struct localeconv	Countries
char *in_curr_symbol	**Italy:** "ITL." **Norway:** "NOK" **Switzerland:** "CHF"
char *currency_symbol	**Italy:** "L." **Norway:** "kr" **Switzerland:** "SFrs."
char *mon_decimal_point	**Italy:** " " **Norway:** "." **Switzerland:** "."
char *mon_thousands_sep	**Italy:** "." **Norway:** "." **Switzerland:** "."
char *mon_grouping	**Italy:** "\3" **Norway:** "\3" **Switzerland:** "\3"

struct localeconv	Countries
char *positive_sign	**Italy:** " " **Norway:** " " **Switzerland:** " "
char *negative_sign	**Italy:** "-" **Norway:** "-" **Switzerland:** "C"
char int_frac_digits	**Italy:** 0 **Norway:** 2 **Switzerland:** 2
char frac_digits	**Italy:** 0 **Norway:** 2 **Switzerland:** 2
char p_cs_precedes	**Italy:** 1 **Norway:** 1 **Switzerland:** 1
char p_sep_by_space	**Italy:** 0 **Norway:** 0 **Switzerland:** 0
char n_cs_precedes	**Italy:** 1 **Norway:** 1 **Switzerland:** 1

struct localeconv	Countries
char n_sep_by_space	**Italy:** 0 **Norway:** 0 **Switzerland:** 0
char p_sign_posn	**Italy:** 1 **Norway:** 1 **Switzerland:** 1
char n_sign_posn	**Italy:** 1 **Norway:** 2 **Switzerland:** 2

Return Values

A pointer to the filled-in object is returned. In addition, calls to the **setlocale** subroutine with the **LC_ALL**, **LC_MONETARY** or **LC_NUMERIC** categories may cause subsequent calls to the **localeconv** subroutine to return different values based on the selection of the locale.

Note: The structure pointed to by the return value is not modified by the program but may be overwritten by a subsequent call to the **localeconv** subroutine.

lockfx, lockf, flock, or lockf64 Subroutine

Purpose

Locks and unlocks sections of open files.

Libraries

lockfx, **lockf**: Standard C Library (**libc.a**)

Item	Description
flock:	Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <fcntl.h>
```

```
int lockfx (FileDescriptor,
Command, Argument)
int  FileDescriptor;
```

```
int  Command;
struct flock * Argument;
```

```
#include <sys/lockf.h>
#include <unistd.h>
```

```
int lockf
(FileDescriptor, Request, Size)
int FileDescriptor;
int  Request;
off_t  Size;
```

```
int lockf64 (FileDescriptor,
Request, Size)
int FileDescriptor;
int  Request;
off64_t  Size;
```

```
#include <sys/file.h>
```

```
int flock (FileDescriptor,  Operation)
int FileDescriptor;
int Operation;
```

Description

Attention: Buffered I/O does not work properly when used with file locking. Do not use the standard I/O package routines on files that are going to be locked.

The **lockfx** subroutine locks and unlocks sections of an open file. The **lockfx** subroutine provides a subset of the locking function provided by the **fcntl** subroutine.

The **lockf** subroutine also locks and unlocks sections of an open file. However, its interface is limited to setting only write (exclusive) locks.

Although the **lockfx**, **lockf**, **flock**, and **fcntl** interfaces are all different, their implementations are fully integrated. Therefore, locks obtained from one subroutine are honored and enforced by any of the lock subroutines.

The *Operation* parameter to the **lockfx** subroutine, which creates the lock, determines whether it is a read lock or a write lock.

The file descriptor on which a write lock is being placed must have been opened with write access.

lockf64 is equivalent to **lockf** except that a 64-bit lock request size can be given. For **lockf**, the largest value which can be used is **OFF_MAX**, for **lockf64**, the largest value is **LONGLONG_MAX**.

In the large file enabled programming environment, **lockf** is redefined to be **lock64**.

The **flock** subroutine locks and unlocks entire files. This is a limited interface maintained for BSD compatibility, although its behavior differs from BSD in a few subtle ways. To apply a shared lock, the file must be opened for reading. To apply an exclusive lock, the file must be opened for writing.

Locks are not inherited. Therefore, a child process cannot unlock a file locked by the parent process.

Parameters

Item	Description
Argument	A pointer to a structure of type **flock**, defined in the **flock.h** file**.**

Item	Description
Command	Specifies one of the following constants for the **lockfx** subroutine: **F_SETLK** Sets or clears a file lock. The `l_type` field of the **flock** structure indicates whether to establish or remove a read or write lock. If a read or write lock cannot be set, the **lockfx** subroutine returns immediately with an error value of -1. **F_SETLKW** Performs the same function as **F_SETLK** unless a read or write lock is blocked by existing locks. In that case, the process sleeps until the section of the file is free to be locked. **F_GETLK** Gets the first lock that blocks the lock described in the **flock** structure. If a lock is found, the retrieved information overwrites the information in the **flock** structure. If no lock is found that would prevent this lock from being created, the structure is passed back unchanged except that the `l_type` field is set to **F_UNLCK**.
FileDescriptor	A file descriptor returned by a successful **open** or **fcntl** subroutine, identifying the file to which the lock is to be applied or removed.
Operation	Specifies one of the following constants for the **flock** subroutine: **LOCK_SH** Apply a shared (read) lock. **LOCK_EX** Apply an exclusive (write) lock. **LOCK_NB** Do not block when locking. This value can be logically ORed with either **LOCK_SH** or **LOCK_EX**. **LOCK_UN** Remove a lock.
Request	Specifies one of the following constants for the **lockf** subroutine: **F_ULOCK** Unlocks a previously locked region in the file. **F_LOCK** Locks the region for exclusive (write) use. This request causes the calling process to sleep if the requested region overlaps a locked region, and to resume when granted the lock. **F_TEST** Tests to see if another process has already locked a region. The **lockf** subroutine returns 0 if the region is unlocked. If the region is locked, then -1 is returned and the **errno** global variable is set to **EACCES**. **F_TLOCK** Locks the region for exclusive use if another process has not already locked the region. If the region has already been locked by another process, the **lockf** subroutine returns a -1 and the **errno** global variable is set to **EACCES**.

Item	Description
Size	The number of bytes to be locked or unlocked for the **lockf** subroutine. The region starts at the current location in the open file, and extends forward if the *Size* value is positive and backward if the *Size* value is negative. If the *Size* value is 0, the region starts at the current location and extends forward to the maximum possible file size, including the unallocated space after the end of the file.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **lockfx**, **lockf**, and **flock** subroutines fail if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid open file descriptor.
EINVAL	The function argument is not one of **F_LOCK**, **F_TLOCK**, **F_TEST** or **F_ULOCK**; or *size* plus the current file offset is less than 0.
EINVAL	An attempt was made to lock a fifo or pipe.
EDEADLK	The lock is blocked by a lock from another process. Putting the calling process to sleep while waiting for the other lock to become free would cause a deadlock.
ENOLCK	The lock table is full. Too many regions are already locked.
EINTR	The command parameter was **F_SETLKW** and the process received a signal while waiting to acquire the lock.
EOVERFLOW	The offset of the first, or if *size* is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type *off_t*.

The **lockfx** and **lockf** subroutines fail if one of the following is true:

Item	Description
EACCES	The *Command* parameter is **F_SETLK**, the `l_type` field is **F_RDLCK**, and the segment of the file to be locked is already write-locked by another process.
EACCES	The *Command* parameter is **F_SETLK**, the `l_type` field is **F_WRLCK**, and the segment of a file to be locked is already read-locked or write-locked by another process.

The **flock** subroutine fails if the following is true:

Item	Description
EWOULDBLOCK	The file is locked and the **LOCK_NB** option was specified.

log10, log10f, log10l, log10d32, log10d64, and log10d128 Subroutine

Purpose

Computes the Base 10 logarithm.

Syntax

```
#include <math.h>

float log10f (x)
float x;

long double log10l (x)
long double x;

double log10 (x)
double x;
_Decimal32 log10d32 (x)
_Decimal32 x;

_Decimal64 log10d64 (x)
_Decimal64 x;

_Decimal128 log10d128 (x)
_Decimal128 x;
```

Description

The **log10f**, **log10l**, **log10**, **log10d32**, **log10d64**, and **log10d128** subroutines compute the base 10 logarithm of the *x* parameter, $\log_{10}(x)$.

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **log10**, **log10f**, **log10l**, **log10d32**, **log10d64**, and **log10d128** subroutines return the base 10 logarithm of *x*.

If *x* is ±0, a pole error occurs and **log10**, **log10f**, **log10l**, **log10d32**, **log10d64**, and **log10d128** subroutines return -**HUGE_VAL**, -**HUGE_VALF**, -**HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

For finite values of *x* that are less than 0, or if *x* is -Inf, a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is 1, +0 is returned.

If *x* is +Inf, +Inf is returned.

Error Codes

When using the **libm.a** library:

Item	Description
log10	If the *x* parameter is less than 0, the **log10** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If x= 0, the **log10** subroutine returns a -**HUGE_VAL** value and sets **errno** to **ERANGE**.

When using **libmsaa.a**(**-lmsaa**):

Item	Description
log10	If the *x* parameter is not positive, the **log10** subroutine returns a **-HUGE_VAL** value and sets **errno** to **EDOM**. A message indicating DOMAIN error (or SING error when *x* = 0) is output to standard error.
log10	If *x* < 0, **log10l** returns the value **NaNQ** and sets **errno** to **EDOM**. If *x* equals 0, **log10l** returns the value **-HUGE_VAL** but does not modify **errno**.

log1p, log1pf, log1pl, log1pd32, log1pd64, and log1pd128 Subroutines

Purpose

Computes a natural logarithm.

Syntax

```
#include <math.h>

float log1pf (x)
float x;

long double log1pl (x)
long double x;

double log1p (x)
double x;
_Decimal32 log1pd32 (x)
_Decimal32 x;

_Decimal64 log1pd64 (x)
_Decimal64 x;

_Decimal128 log1pd128 (x)
_Decimal128 x;
```

Description

The **log1pf**, **log1pl**, **log1p**, **log1pd32**, **log1pd64**, and **log1pd128** subroutines compute $\log_e (1.0 + x)$.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **log1pf**, **log1pl**, **log1p**, **log1pd32**, **log1pd64**, and **log1pd128** subroutines return the natural logarithm of 1.0 + *x*.

If *x* is -1, a pole error occurs and the **log1pf**, **log1pl**, **log1p**, **log1pd32**, **log1pd64**, and **log1pd128** subroutines return **-HUGE_VALF**, **-HUGE_VALL**, **-HUGE_VAL**, **-HUGE_VAL_D32**, **-HUGE_VAL_D64**, and **-HUGE_VAL_D128** respectively.

For finite values of *x* that are less than -1, or if *x* is -Inf, a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0, or +Inf, *x* is returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

log2, log2f, log2l, log2d32, log2d64, and log2d128 Subroutine

Purpose

Computes base 2 logarithm.

Syntax

```
#include <math.h>

double log2 (x)
double x;

float log2f (x)
float x;

long double log2l (x)
long double x;
_Decimal32 log2d32 (x)
_Decimal32 x;

_Decimal64 log2d64 (x)
_Decimal64 x;

_Decimal128 log2d128 (x)
_Decimal128 x;
```

Description

The **log2**, **log2f**, **log2l**, **log2d32**, **log2d64**, and **log2d128** subroutines compute the base 2 logarithm of the *x* parameter, $\log_2 (x)$.

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **log2**, **log2f**, **log2l**, **log2d32**, **log2d64**, and **log2d128** subroutines return the base 2 logarithm of *x*.

If *x* is ±0, a pole error occurs and the **log2**, **log2f**, **log2l**, **log2d32**, **log2d64**, and **log2d128** subroutines return -**HUGE_VAL**, -**HUGE_VALF**, -**HUGE_VALL**, -**HUGE_VAL_D32**, -**HUGE_VAL_D64**, and -**HUGE_VAL_D128** respectively.

For finite values of *x* that are less than 0, or if *x* is -Inf, a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is 1, +0 is returned.

If *x* is +Inf, *x* is returned.

logbd32, logbd64, and logbd128 Subroutines

Purpose

Computes the radix-independent exponent.

Syntax

```
#include <math.h>

_Decimal32 logbd32 (x)
_Decimal32 x;

_Decimal64 logbd64 (x)
_Decimal64 x;

_Decimal128 logbd128 (x)
_Decimal128 x;
```

Description

The **logbd32**, **logbd64**, and **logbd128** subroutines compute the exponent of *x*, which is an integral part of $\log_r |x|$, as a signed floating-point value, for nonzero *x*. In the $\log_r |x|$, the *r* is the radix of the machine's decimal floating-point arithmetic. For AIX, FLT_RADIX r=10.

An application that wants to check for error situations must set the **errno** to zero and call the **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. On return, if the **errno** is of the value of nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is of the value of nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **logbd32**, **logbd64**, and **logbd128** subroutines return the exponent of *x*.

If *x* is ±0, a pole error occurs and the **logbd32**, **logbd64**, and **logbd128** subroutines return -**HUGE_VAL_D32**, -**HUGE_VAL_D64**, and -**HUGE_VAL_D128**, respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±Inf, +Inf is returned.

logbf, logbl, or logb Subroutine

Purpose

Computes the radix-independent exponent.

Syntax

```
#include <math.h>

float logbf (x)
float x;

long double logbl (x)
long double x;
```

```
double logb(x)
double x;
```

Description

The **logbf** and **logbl** subroutines compute the exponent of *x*, which is the integral part of $\log_r |x|$, as a signed floating-point value, for nonzero *x*, where *r* is the radix of the machine's floating-point arithmetic. For AIX, FLT_RADIX r=2.

If *x* is subnormal, it is treated as though it were normalized; thus for finite positive *x*:

```
1 <= x * FLT_RADIX^-logb(x) < FLT_RADIX
```

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Note: When the *x* parameter is finite and not zero, the **logb** *(x)* subroutine satisfies the following equation:

```
1 < = scalb (|x|, -(int) logb (x)) < 2
```

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **logbf** and **logbl** subroutines return the exponent of *x*.

If *x* is ±0, a pole error occurs and the **logbf** and **logbl** subroutines return -**HUGE_VALF** and -**HUGE_VALL**, respectively.

If *x* is NaN, a NaN is returned.

If *x* is ±Inf, +Inf is returned.

Error Codes

The **logb** function returns **-HUGE_VAL** when the x parameter is set to a value of 0 and sets **errno** to **EDOM**.

log, logf, logl, logd32, logd64, and logd128 Subroutines

Purpose

Computes the natural logarithm.

Syntax

```
#include <math.h>

float logf (x)
float x;

long double logl (x)
long double x;

double log (x)
double x;
```

```
_Decimal32 logd32 (x)
_Decimal32 x;

_Decimal64 logd64 (x)
_Decimal64 x;

_Decimal128 logd128 (x)
_Decimal128 x;
```

Description

The **logf**, **logl**, **log**, **logd32**, **logd64**, and **logd128** subroutines compute the natural logarithm of the *x* parameter, $\log_e(x)$.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **logf**, **logl**, **log**, **logd32**, **logd64**, and **logd128** subroutines return the natural logarithm of *x*.

If *x* is ±0, a pole error occurs and the **logf**, **logl**, and **log** subroutines return -**HUGE_VALF** and -**HUGE_VALL**, -**HUGE_VAL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

For finite values of *x* that are less than 0, or if *x* is -Inf, a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is 1, +0 is returned.

If *x* is +Inf, *x* is returned.

Error Codes

When using the **libm.a** library:

Item	Description
log	If the *x* parameter is less than 0, the **log** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If x= 0, the **log** subroutine returns a -**HUGE_VAL** value but does not modify **errno**.

When using **libmsaa.a**(**-lmsaa**):

Item	Description
log	If the *x* parameter is not positive, the **log** subroutine returns a -**HUGE_VAL** value, and sets **errno** to a **EDOM** value. A message indicating DOMAIN error (or SING error when *x* = 0) is output to standard error.
log	If *x*<0, the **logl** subroutine returns a **NaNQ** value

loginfailed Subroutine

Purpose

Records an unsuccessful login attempt.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int loginfailed ( User,  Host,  Tty, Reason)
char *User;
char *Host;
char *Tty;
int Reason;
```

Note: This subroutine is not thread-safe.

Description

The **loginfailed** subroutine performs the processing necessary when an unsuccessful login attempt occurs. If the specified user name is not valid, the **UNKNOWN_USER** value is substituted for the user name. This substitution prevents passwords entered as the user name from appearing on screen.

The following attributes in **/etc/security/lastlog** file are updated for the specified user, if the user name is valid:

Item	Description
time_last_unsuccessful_login	Contains the current time.
tty_last_unsuccessful_login	Contains the value specified by the *Tty* parameter.
host_last_unsuccessful_login	Contains the value specified by the *Host* parameter, or the local hostname if the *Host* parameter is a null value.
unsuccessful_login_count	Indicates the number of unsuccessful login attempts. The **loginfailed** subroutine increments this attribute by one for each failed attempt.

A login failure audit record is cut to indicate that an unsuccessful login attempt occurred. A **utmp** entry is appended to **/etc/security/failedlogin** file, which tracks all failed login attempts.

If the current unsuccessful login and the previously recorded unsuccessful logins constitute too many unsuccessful login attempts within too short of a time period (as specified by the **logindisable** and **logininterval** port attributes), the port is locked. When a port is locked, a PORT_Locked audit record is written to inform the system administrator that the port has been locked.

If the login retry delay is enabled (as specified by the **logindelay** port attribute), a sleep occurs before this subroutine returns. The length of the sleep (in seconds) is determined by the **logindelay** value multiplied by the number of unsuccessful login attempts that occurred in this process.

Parameters

Item	Description
User	Specifies the user's login name who has unsuccessfully attempted to login.

Item	Description
Host	Specifies the name of the host from which the user attempted to login. If the *Host* parameter is Null, the name of the local host is used.
Tty	Specifies the name of the terminal on which the user attempted to login.
Reason	Specifies a reason code for the login failure. Valid values are AUDIT_FAIL and AUDIT_FAIL_AUTH defined in the **sys/audit.h** file.

Security

Access Control: The calling process must have access to the account information in the user database and the port information in the port database.

File Accessed:

Mode	File
r	/etc/security/user
rw	/etc/security/lastlog
r	/etc/security/login.cfg
rw	/etc/security/portlog
w	/etc/security/failedlogin

Auditing Events:

Event	Information
USER_Login	username
PORT_Locked	portname

Return Values

Upon successful completion, the **loginfailed** subroutine returns a value of 0. If an error occurs, a value of -1 is returned and errno is set to indicate the error.

Error Codes

The **loginfailed** subroutine fails if one or more of the following values is true:

Item	Description
EACCES	The current process does not have access to the user or port database.
EPERM	The current process does not have permission to write an audit record.

loginrestrictions Subroutine

Purpose

Determines if a user is allowed to access the system.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <login.h>
```

```
int loginrestrictions (Name, Mode, Tty, Msg)
char * Name;
int  Mode;
char * Tty;
char ** Msg;
```

Note: This subroutine is not thread-safe.

Description

The **loginrestrictions** subroutine determines if the user specified by the *Name* parameter is allowed to access the system. The *Mode* parameter gives the mode of account usage and the *Tty* parameter defines the terminal used for access. The *Msg* parameter returns an informational message explaining why the **loginrestrictions** subroutine failed.

This subroutine is unsuccessful if any of the following conditions exists:

- The user's account has expired as defined by the **expires** user attribute.
- The user's account has been locked as defined by the **account_locked** user attribute.
- The user attempted too many unsuccessful logins as defined by the **loginretries** user attribute.
- The user is not allowed to access the given terminal as defined by the **ttys** user attribute.
- The user is not allowed to access the system at the present time as defined by the **logintimes** user attribute.
- The *Mode* parameter is set to the **S_LOGIN** value or the **S_RLOGIN** value, and too many users are logged in as defined by the **maxlogins** system attribute.
- The *Mode* parameter is set to the **S_LOGIN** value and the user is not allowed to log in as defined by the **login** user attribute.
- The *Mode* parameter is set to the **S_RLOGIN** value and the user is not allowed to log in from the network as defined by the **rlogin** user attribute.
- The *Mode* parameter is set to the **S_SU** value and other users are not allowed to use the **su** command as defined by the **su** user attribute, or the group ID of the current process cannot use the **su** command to switch to this user as defined by the **sugroups** user attribute.
- The *Mode* parameter is set to the **S_DAEMON** value and the user is not allowed to run processes from the **cron** or **src** subsystem as defined by the **daemon** user attribute.
- The terminal is locked as defined by the **locktime** port attribute.
- The user cannot use the terminal to access the system at the present time as defined by the **logintimes** port attribute.
- The user is not the root user and the **/etc/nologin** file exists.

Note: The **loginrestrictions** subroutine is not safe in a multi-threaded environment. To use **loginrestrictions** in a threaded application, the application must keep the integrity of each thread.

Parameters

Item	Description
Name	Specifies the user's login name whose account is to be validated.

Item	Description
Mode	Specifies the mode of usage. Valid values as defined in the **login.h** file are listed below. The *Mode* parameter has a value of 0 or one of the following values: **S_LOGIN** Verifies that local logins are permitted for this account. **S_SU** Verifies that the **su** command is permitted and the current process has a group ID that can invoke the **su** command to switch to the account. **S_DAEMON** Verifies the account can invoke daemon or batch programs through the **src** or **cron** subsystems. **S_RLOGIN** Verifies the account can be used for remote logins through the **rlogind** or **telnetd** programs.
Tty	Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no tty origin checking is done.
Msg	Returns an informative message indicating why the **loginrestrictions** subroutine failed. Upon return, the value is either a pointer to a valid string within memory allocated storage or a null value. If a message is displayed, it is provided based on the user interface.

Security

Access Control:The calling process must have access to the account information in the user database and the port information in the port database.

File Accessed:

Mode	Files
r	/etc/security/user
r	/etc/security/login.cfg
r	/etc/security/portlog
r	/etc/passwd

Return Values

If the account is valid for the specified usage, the **loginrestrictions** subroutine returns a value of 0. Otherwise, a value of -1 is returned, the **errno** global value is set to the appropriate error code, and the *Msg* parameter returns an informative message explaining why the specified account usage is invalid.

Error Codes

The **loginrestrictions** subroutine fails if one or more of the following values is true:

Item	Description
ENOENT	The user specified does not have an account.
ESTALE	The user's account is expired.
EPERM	The user's account is locked, the specified terminal is locked, the user has had too many unsuccessful login attempts, or the user cannot log in because the **/etc/nologin** file exists.

Item	Description
EACCES	One of the following conditions exists: • The specified terminal does not have access to the specified account. • The *Mode* parameter is the **S_SU** value and the current process is not permitted to use the **su** command to access the specified user. • Access to the account is not permitted in the specified mode. • Access to the account is not permitted at the current time. • Access to the system with the specified terminal is not permitted at the current time.
EAGAIN	The *Mode* parameter is either the **S_LOGIN** value or the **S_RLOGIN** value, and all the user licenses are in use.
EINVAL	The *Mode* parameter has a value other than **S_LOGIN**, **S_SU**, **S_DAEMON**, **S_RLOGIN**, or **0**.

loginrestrictionsx Subroutine

Purpose

Determines, in multiple methods, if a user is allowed to access the system.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <login.h>

int loginrestrictionsx (Name, Mode, Tty, Message, State)
char * Name;
int  Mode;
char *Tty;
char **Message;
void **State;
```

Description

The **loginrestrictionsx** subroutine determines if the user specified by the *Name* parameter is allowed to access the system. The *Mode* parameter gives the mode of account usage, and the *Tty* parameter defines the terminal used for access. The *Msg* parameter returns an informational message explaining why the **loginrestrictionsx** subroutine failed. The user's **SYSTEM** attribute determines the administrative domains to examine for permission.

The *State* parameter contains information about the login restrictions for the user. A call to the **authenticatex** subroutine will not use an administrative domain for authentication if an earlier call to **loginrestrictionsx** indicated that the user was unable to log in using that administrative domain's authentication data. The result is that administrative domains that are used for authentication must permit the user to log in. The *State* parameter returned by **loginrestrictionsx** can be used as input to a subsequent call to the **authenticatex** subroutine.

This subroutine is unsuccessful if any of the following conditions exists:

- The user's account has been locked as defined by the **account_locked** user attribute.
- The user's account has expired as defined by the **expires** user attribute.

- The *Mode* parameter is set to the **S_LOGIN** value or the **S_RLOGIN** value, and too many users are logged in as defined by the **maxlogins** system attribute.
- The *Mode* parameter is not set to the **S_SU** or **S_DAEMON** value, and the user is not allowed to log in to the current host as defined by the user's **hostallowedlogin** and **hostdeniedlogin** attributes.
- The user is not allowed to access the system at the present time as defined by the **logintimes** user attribute.
- The user attempted too many unsuccessful logins as defined by the **loginretries** user attribute.
- The user is not allowed to access the given terminal or network protocol as defined by the **ttys** user attribute. This test is not performed when the *Mode* parameter is set to the **S_DAEMON** value.
- The *Mode* parameter is set to the **S_LOGIN** value, and the user is not allowed to log in as defined by the **login** user attribute.
- The *Mode* parameter is set to the **S_RLOGIN** value and the user is not allowed to log in from the network as defined by the **rlogin** user attribute.
- The Mode parameter is set to the **S_SU** value, and other users are not allowed to use the **su** command as defined by the **su** user attribute; or, the group ID of the current process cannot use the **su** command to switch to this user as defined by the **sugroups** user attribute.
- The *Mode* parameter is set to the **S_DAEMON** value, and the user is not allowed to run processes from the **cron** or **src** subsystem as defined by the **daemon** user attribute.
- The terminal is locked as defined by the **locktime** port attribute.
- The user cannot use the terminal to access the system at the present time as defined by the **logintimes** port attribute.
- The user is not the root user, and the **/etc/nologin** file exists.

Additional restrictions can be enforced by loadable authentication modules for any administrative domain used in the user's **SYSTEM** attribute.

Parameters

Item	Description
Name	Specifies the user's login name whose account is to be validated.
Mode	Specifies the mode of usage. The valid values in the following list are defined in the **login.h** file. The *Mode* parameter has a value of 0 or one of the following values: **S_LOGIN** Verifies that local logins are permitted for this account. **S_SU** Verifies that the **su** command is permitted and the current process has a group ID that can invoke the **su** command to switch to the account. **S_DAEMON** Verifies that the account can invoke daemon or batch programs through the **src** or **cron** subsystems. **S_RLOGIN** Verifies that the account can be used for remote logins through the **rlogind** or **telnetd** programs.
Tty	Specifies the terminal of the originating activity. If this parameter is a null pointer or a null string, no **tty** origin checking is done. The *Tty* parameter can also have the value **RSH** or **REXEC** to indicate that the caller is the **rsh** or **rexec** command.
Message	Returns an informative message indicating why the **loginrestrictionsx** subroutine failed. Upon return, the value is either a pointer to a valid string within memory-allocated storage or a null value. If a message is displayed, it is provided based on the user interface.

Item	Description
State	Points to a pointer that the **loginrestrictionsx** subroutine allocates memory for and fills in. The *State* parameter can also be the result of an earlier call to the **authenticatex** subroutine. The *State* parameter contains information about the results of the **loginrestrictionsx** subroutine for each term in the user's **SYSTEM** attribute. The calling application is responsible for freeing this memory when it is no longer needed for a subsequent call to the **authenticatex**, **passwdexpiredx**, or **chpassx** subroutines.

Security

Access Control: The calling process must have access to the account information in the user database and the port information in the port database.

Files accessed:

Item	Description
Mode	**File**
r	**/etc/security/user**
r	**/etc/security/login.cfg**
r	**/etc/security/portlog**
r	**/etc/passwd**

Return Values

If the account is valid for the specified usage, the **loginrestrictionsx** subroutine returns a value of 0. Otherwise, a value of -1 is returned, the **errno** global value is set to the appropriate error code, and the *Message* parameter returns an informative message explaining why the specified account usage is invalid.

Error Codes

If the **loginrestrictionsx** subroutine fails if one of the following values is true:

Item	Description
EACCES	One of the following conditions exists: • The specified terminal does not have access to the specified account. • The *Mode* parameter is the **S_SU** value, and the current process is not permitted to use the **su** command to access the specified user. • Access to the account is not permitted in the specified mode. • Access to the account is not permitted at the current time. • Access to the system with the specified terminal is not permitted at the current time.
EAGAIN	The *Mode* parameter is either the **S_LOGIN** value or the **S_RLOGIN** value, and all the user licenses are in use.
EINVAL	The *Mode* parameter has a value other than **S_LOGIN**, **S_SU**, **S_DAEMON**, **S_RLOGIN**, or 0.
ENOENT	The user specified does not have an account.
EPERM	The user's account is locked, the specified terminal is locked, the user has had too many unsuccessful login attempts, or the user cannot log in because the **/etc/nologin** file exists.

Item	Description
ESTALE	The user's account is expired.

loginsuccess Subroutine

Purpose

Records a successful log in.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int loginsuccess (User, Host, Tty, Msg)
char * User;
char * Host;
char * Tty;
char ** Msg;
```

Note: This subroutine is not thread-safe.

Description

The **loginsuccess** subroutine performs the processing necessary when a user successfully logs into the system. This subroutine updates the following attributes in the **/etc/security/lastlog** file for the specified user:

Item	Description
time_last_login	Contains the current time.
tty_last_login	Contains the value specified by the *Tty* parameter.
host_last_login	Contains the value specified by the *Host* parameter or the local host name if the *Host* parameter is a null value.
unsuccessful_login_count	Indicates the number of unsuccessful login attempts. The **loginsuccess** subroutine resets this attribute to a value of 0.

Additionally, a login success audit record is cut to indicate in the audit trail that this user has successfully logged in.

A message is returned in the *Msg* parameter that indicates the time, host, and port of the last successful and unsuccessful login. The number of unsuccessful login attempts since the last successful login is also provided to the user.

Parameters

Item	Description
User	Specifies the login name of the user who has successfully logged in.
Host	Specifies the name of the host from which the user logged in. If the *Host* parameter is a null value, the name of the local host is used.
Tty	Specifies the name of the terminal which the user used to log in.

Item	Description
Msg	Returns a message indicating the delete time, host, and port of the last successful and unsuccessful logins. The number of unsuccessful login attempts since the last successful login is also provided. Upon return, the value is either a pointer to a valid string within memory allocated storage or a null pointer. It is the responsibility of the calling program to **free( )** the returned storage.

Security

Access Control: The calling process must have access to the account information in the user database.

File Accessed:

Mode	File
rw	/etc/security/lastlog

Auditing Events:

Event	Information
USER_Login	username

Return Values

Upon successful completion, the **loginsuccess** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Note: If the load module does not have interface support that is defined in the security library, the **loginsuccess** subroutine might return a value of 0 (success), and display ENOSYS as the **errno** value.

Error Codes

The **loginsuccess** subroutine fails if one or more of the following values is true:

Item	Description
ENOENT	The specified user does not exist.
EACCES	The current process does not have write access to the user database.
EPERM	The current process does not have permission to write an audit record.
ENOSYS	The load module does not have the required interface support defined in the security library.

lpar_get_info Subroutine

Purpose

Retrieves the characteristics of the calling partition.

Syntax

```
#include <sys/dr.h>

int lpar_get_info (command, lparinfo, bufsize)
int command;
void *lparinfo;
size_t bufsize;
```

Description

The **lpar_get_info** subroutine retrieves processor module information, and both LPAR and Micro-Partitioning® attributes of low-frequency use and high-frequency use. Because the low-frequency attributes, as defined in the **lpar_info_format1_t** structure, are static in nature, a reboot is required to effect any change. The high-frequency attributes, as defined in the **lpar_info_format2_t** structure, can be changed dynamically at any time either by the platform or through dynamic logical partitioning (DLPAR) procedures. The latter provides a mechanism for notifying applications of changes. The signature of this system call, its parameter types, and the order of the member fields in both the **lpar_info_format1_t** and **lpar_info_format2_t** structures are specific to the AIX platform. If the **WPAR_INFO_FORMAT** command is specified, the WPAR attributes are returned in a **wpar_info_format_t** structure. To request processor module information, specify the **PROC_MODULE_INFO** command. The information is provided as an array of **proc_module_info_t** structures. To obtain this information, you must provide a buffer of exact length to accommodate one **proc_module_info_t** structure for each module type. The module count can be obtained by using the **NUM_PROC_MODULE_TYPES** command, and it is in the form of a **uint64_t** type. Processor module information is reported for the entire system. This information is available on POWER6® and later systems.

To see the complete structures of **lpar_info_format1_t**, **lpar_info_format2_t**, **wpar_info_format_t**, and **proc_module_info_t**, see the **dr.h** header file.

The **lpar_get_info** system call provides information about the operating system environment, including the following:

- Type of partition: dedicated processor partition or micro-partition
- Type of micro-partition: capped or uncapped
- Variable capacity weight of micro-partition
- Partition name and number
- SMT-capable partition
- SMT-enabled partition
- Minimum, desired, online, and maximum number of virtual processors
- Minimum, online, and maximum number of logical processors
- Minimum, desired, online, and maximum entitled processor capacity
- Minimum, desired, online (megabytes), and maximum number of logical memory blocks (LMBs)
- Maximum number of potential installed physical processors in the server, including unlicensed and potentially hot-pluggable
- Number of active licensed installed physical processors in the server
- Number of processors in the shared processor pool
- Workload partition static identifier
- Workload partition dynamic identifier
- Workload partition processor limits
- Socket, chip, and core topology of the system that the processor module information provides
- Logical pages coalesced in active memory sharing enabled partitions.
- Physical pages coalesced in memory pools in active memory sharing enabled partitions.
- PURR and SPURR consumed for page coalescing in active memory sharing enabled partitions.

This subroutine is used by the DRM to determine whether a client partition is migration capable and MSP capable. The kernel presents these capabilities based on the presence of the **hcall-vasi** function set and the type of partition that is evident. If the partition is a VIOS partition, the MSP capability will be noted. Otherwise, the OS partition migration capability will be noted.

Parameters

Item	Description
command	Specifies whether the user wants **format1**, **format2**, workload partition, or processor module details.
lparinfo	Pointer to the user-allocated buffer that is passed in.
bufsize	Size of the buffer that is passed in.

Return Values

Upon success, the **lpar_get_info** subroutine returns a value of 0. Upon failure, a value of -1 is returned, and **errno** is set to indicate the appropriate error.

Error Codes

Item	Description
EFAULT	Buffer size is smaller than expected.
EINVAL	Invalid input parameter.
ENOSYS	The hardware or the current firmware level does not support this operation.
ENOTSUP	The platform does not support this operation.

Example

The following example demonstrates how to retrieve processor module information using the **lpar_get_info** subroutine:

```
uint64_t             module_count;
proc_module_info_t   *buffer = NULL;
int                  rc = 0;

/* Retrieve the total count of modules on the system */
rc = lpar_get_info(NUM_PROC_MODULE_TYPES,
                   &module_count, sizeof(uint64_t));

if (rc)
    return(1);  /* Error */

/* Allocate buffer of exact size to accomodate module information */
buffer = malloc(module_count * sizeof(proc_module_info_t));

if (buffer == NULL)
    return(2);

rc = lpar_get_info(PROC_MODULE_INFO, buffer, (module_count * sizeof(proc_module_info_t)));

if (rc)
    return(3);  /* Error */

/*  If rc is 0, then buffer contains an array of proc_module_info_t
 *  structures with module_count elements.  For an element of
 *  index i:
 *
 *      buffer[i].nsockets is the total number of sockets
 *      buffer[i].nchips   is the number of chips per socket
 *      buffer[i].ncores   is the number of cores per chip
 */
```

lpar_set_resources Subroutine

Purpose

Modifies the calling partition's characteristics.

Library

Standard C Library (`lib.c`)

Syntax

```
#include  <sys/dr.h>
```

```
int lpar_set_resources ( lpar_resource_id,lpar_resource )
int lpar_resource_id;
void *lpar_resource;
```

Description

The `lpar_set_resources` subroutine modifies the configuration attributes (dynamic resources) on a current partition indicated by the *lpar_resource_id*. The pointer to a value of the dynamic resource indicated by *lpar_resource_id* is passed to this call in *lpar_resource*. This subroutine modifies one partition dynamic resource at a time. To reconfigure multiple resources, multiple calls must be made. The following resources for the calling partition can be modified:

- Processor Entitled Capacity
- Processor Variable Capacity Weight
- Number of online virtual processors
- Number of available memory in megabytes
- I/O Entitled Memory Capacity in bytes
- Variable Memory Capacity Weight

These resource IDs are defined in the <**sys/dr.h**> header file. To modify the Processor Entitled Capacity and Processor Variable Capacity Weight attributes, ensure that the current partition is an SPLPAR partition. Otherwise, an error is returned.

Note: The **lpar_set_resources** subroutine can only be called in a process owned by a root user or a user with the CAP_EWLM_AGENT capability. Otherwise, an error is returned.

Parameters

Item	Description
lpar_resource_id	Identifies the dynamic resource whose value is being changed.
lpar_resource	Pointer to a new value of the dynamic resource identified by the *lpar_resource_id*.

Security

The `lpar_set_resources` subroutine can only be called in a process owned by a root user (super user) or a user with the `CAP_EWLM_AGENT` capability.

Return Values

Upon success, the `lpar_set_resources` subroutine returns a value of `0`. Upon failure, a negative value is returned, and `errno` is set to the appropriate error.

Error Codes

Item	Description
EINVAL	Invalid configuration parameters.
EPERM	Insufficient authority.
EEXIST	Resource already exists.
EBUSY	Resource is busy.
EAGAIN	Resource is temporarily unavailable.
ENOMEM	Resource allocation failed.
ENOTREADY	Resource is not ready.
ENOTSUP	Operation is not supported.
EFAULT/EIO	Operation failed because of an I/O error.
EINPROGRESS	Operation in progress.
ENXIO	Resource is not available.
ERANGE	Parameter value is out of range.
All others	Internal error.

lrint, lrintf, lrintl, lrintd32, lrintd64, and lrintd128 Subroutines

Purpose

Round to nearest integer value using the current rounding direction.

Syntax

```
#include <math.h>

long lrint (x)
double x;

long lrintf (x)
float x;

long lrintl (x)
long double x;

long lrintd32 (x)
_Decimal32 x;

long lrintd64 (x)
_Decimal64 x;

long lrintd128 (x)
_Decimal128 x;
```

Description

The **lrint**, **lrintf**, **lrintl**, **lrintd32**, **lrintd64**, and **lrintd128** subroutines round the *x* parameter to the nearest integer value, rounding according to the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be rounded.

Return Values

Upon successful completion, the **lrint**, **lrintf**, **lrintl**, **lrintd32**, **lrintd64**, and **lrintd128** subroutines return the rounded integer value.

If *x* is NaN, a domain error occurs and an unspecified value is returned.

If *x* is +Inf, a domain error occurs and an unspecified value is returned.

If *x* is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a long, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a long, a domain error occurs and an unspecified value is returned.

lround, lroundf, lroundl, lroundd32, lroundd64, and lroundd128 Subroutines

Purpose

Rounds to the nearest integer value.

Syntax

```
#include <math.h>

long lround (x)
double x;

long lroundf (x)
float x;

long lroundl (x)
long double x;

long lroundd32(x)
_Decimal32 x;

long lroundd64(x)
_Decimal64 x;

long lroundd128(x)
_Decimal128 x;
```

Description

The **lround**, **lroundf**, **lroundl**, **lroundd32**, **lroundd64**, and **lroundd128** subroutines round the *x* parameter to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be rounded.

Return Values

Upon successful completion, the **lround**, **lroundf**, **lroundl**, **lroundd32**, **lroundd64**, and **lroundd128** subroutines return the rounded integer value.

If *x* is NaN, a domain error occurs and an unspecified value is returned.

If *x* is +Inf, a domain error occurs and an unspecified value is returned.

If *x* is -Inf, a domain error occurs and an unspecified value is returned.

If the correct value is positive and too large to represent as a **long**, a domain error occurs and an unspecified value is returned.

If the correct value is negative and too large to represent as a **long**, a domain error occurs and an unspecified value is returned.

lsearch or lfind Subroutine

Purpose

Performs a linear search and update.

Library

Standard C Library (**libc.a**)

Syntax

```
void *lsearch (Key, Base, NumberOfElementsPointer, Width, ComparisonPointer)
const void *Key;
void *Base;
size_t Width, *NumberOfElementsPointer;
int (*ComparisonPointer) (cont void*, const void*);
```

```
void *lfind (Key, Base, NumberOfElementsPointer, Width, ComparisonPointer)
const void *Key, Base;
size_t Width, *NumberOfElementsPointer;
int (*ComparisonPointer) (cont void*, const void*);
```

Description

Warning: Undefined results can occur if there is not enough room in the table for the **lsearch** subroutine to add a new item.

The **lsearch** subroutine performs a linear search.

The algorithm returns a pointer to a table where data can be found. If the data is not in the table, the program adds it at the end of the table.

The **lfind** subroutine is identical to the **lsearch** subroutine, except that if the data is not found, it is not added to the table. In this case, a NULL pointer is returned.

The pointers to the *Key* parameter and the element at the base of the table should be of type pointer-to-element and cast to type pointer-to-character. The value returned should be cast into type pointer-to-element.

The comparison function need not compare every byte; therefore, the elements can contain arbitrary data in addition to the values being compared.

Parameters

Item	Description
Base	Points to the first element in the table.
ComparisonPointer	Specifies the name (that you supply) of the comparison function (**strcmp**, for example). It is called with two parameters that point to the elements being compared.
Key	Specifies the data to be sought in the table.
NumberOfElementsPointer	Points to an integer containing the current number of elements in the table. This integer is incremented if the data is added to the table.
Width	Specifies the size of an element in bytes.

The comparison function compares its parameters and returns a value as follows:

- If the first parameter equals the second parameter, the *ComparisonPointer* parameter returns a value of 0.
- If the first parameter does not equal the second parameter, the *ComparisonPointer* parameter returns a value of 1.

Return Values

If the sought entry is found, both the **lsearch** and **lfind** subroutines return a pointer to it. Otherwise, the **lfind** subroutine returns a null pointer and the **lsearch** subroutine returns a pointer to the newly added element.

lseek, llseek or lseek64 Subroutine

Purpose

Moves the read-write file pointer.

Library

Standard C Library (**libc.a**)

Syntax

```
off_t lseek ( FileDescriptor,  Offset,  Whence)
int FileDescriptor, Whence;
off_t Offset;
```

```
offset_t llseek (FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
offset_t Offset;
```

```
off64_t lseek64 (FileDescriptor, Offset, Whence)
int FileDescriptor, Whence;
off64_t Offset;
```

Description

The **lseek**, **llseek**, and **lseek64** subroutines set the read-write file pointer for the open file specified by the *FileDescriptor* parameter. The **lseek** subroutine limits the *Offset* to **OFF_MAX**.

In the large file enabled programming environment, **lseek** subroutine is redefined to **lseek64**.

If the *FileDescriptor* parameter refers to a shared memory object, the **lseek** subroutine fails with **EINVAL**.

Parameters

Item	Description
FileDescriptor	Specifies a file descriptor obtained from a successful **open** or **fcntl** subroutine.
Offset	Specifies a value, in bytes, that is used in conjunction with the *Whence* parameter to set the file pointer. A negative value causes seeking in the reverse direction.
Whence	Specifies how to interpret the *Offset* parameter by setting the file pointer associated with the *FileDescriptor* parameter to one of the following variables: **SEEK_SET** Sets the file pointer to the value of the *Offset* parameter. **SEEK_CUR** Sets the file pointer to its current location plus the value of the *Offset* parameter. **SEEK_END** Sets the file pointer to the size of the file plus the value of the *Offset* parameter.

Return Values

Upon successful completion, the resulting pointer location, measured in bytes from the beginning of the file, is returned. If either the **lseek** or **llseek** subroutines are unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **lseek** or **llseek** subroutines are unsuccessful and the file pointer remains unchanged if any of the following are true:

Item	Description
EBADF	The *FileDescriptor* parameter is not an open file descriptor.
EINVAL	The resulting offset would be greater than the maximum offset allowed for the file or device associated with *FileDescriptor*. The **lseek** subroutine was used with a file descriptor obtained from a call to the **shm_open** subroutine.
EINVAL	Whence is not one of the supported values.
EOVERFLOW	The resulting offset is larger than can be returned properly.
ESPIPE	The *FileDescriptor* parameter is associated with a pipe (FIFO) or a socket.

Files

Item	Description
/usr/include/unistd.h	Defines standard macros, data types and subroutines.

lvm_querylv Subroutine

Purpose

Queries a logical volume and returns all pertinent information.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>
```

```
int lvm_querylv ( LV_ID,  QueryLV,  PVName)
struct lv_id *LV_ID;
struct querylv **QueryLV;
char *PVName;
```

Description

Note: The **lvm_querylv** subroutine uses the **sysconfig** system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the **lvm_querylv** subroutine.

The **lvm_querylv** subroutine returns information for the logical volume specified by the *LV_ID* parameter.

The **querylv** structure, found in the **lvm.h** file, is defined as follows:

```
struct querylv {
      char lvname[LVM_NAMESIZ];
      struct unique_id vg_id;
      long maxsize;
      long mirror_policy;
      long lv_state;
      long currentsize;
      long ppsize;
      long permissions;
      long bb_relocation;
      long write_verify;
      long mirwrt_consist;
      long open_close;
      struct pp *mirrors[LVM_NUMCOPIES];
      unsigned int     stripe_exp;
      unsigned int     striping_width;
}
struct pp {
      struct unique_id pv_id;
      long lp_num;
      long pp_num;
      long ppstate;
 }
```

Field	Description
lvname	Specifies the special file name of the logical volume and can be either the full path name or a single file name that must reside in the **/dev** directory (for example, **rhd1**). All name fields must be null-terminated strings of from 1 to **LVM_NAMESIZ** bytes, including the null byte. If a raw or character device is not specified for the **lvname** field, the Logical Volume Manager (LVM) will add an **r** to the file name to have a raw device name. If there is no raw device entry for this name, the LVM will return the **LVM_NOTCHARDEV** error code.
vg_id	Specifies the unique ID of the volume group that contains the logical volume.

Field	Description
maxsize	Indicates the maximum size in logical partitions for the logical volume and must be in the range of 1 to **LVM_MAXLPS**.
mirror_policy	Specifies how the physical copies are written. The **mirror_policy** field should be either **LVM_SEQUENTIAL** or **LVM_PARALLEL** to indicate how the physical copies of a logical partition are to be written when there is more than one copy.
lv_state	Specifies the current state of the logical volume and can have any of the following bit-specific values ORed together: **LVM_LVDEFINED** The logical volume is defined. **LVM_LVSTALE** The logical volume contains stale partitions.
currentsize	Indicates the current size in logical partitions of the logical volume. The size, in bytes, of every physical partition is 2 to the power of the **ppsize** field.
ppsize	Specifies the size of the physical partitions of all physical volumes in the volume group.
permissions	Specifies the permission assigned to the logical volume and can be one of the following values: **LVM_RDONLY** Access to this logical volume is read only. **LVM_RDWR** Access to this logical volume is read/write.
bb_relocation	Specifies if bad block relocation is desired and is one of the following values: **LVM_NORELOC** Bad blocks will not be relocated. **LVM_RELOC** Bad blocks will be relocated.
write_verify	Specifies if write verification for the logical volume is desired and returns one of the following values: **LVM_NOVERIFY** Write verification is not performed for this logical volume. **LVM_VERIFY** Write verification is performed on all writes to the logical volume.

Field	Description
mirwrt_consist	Indicates whether mirror-write consistency recovery will be performed for this logical volume. The LVM always ensures data consistency among mirrored copies of a logical volume during normal I/O processing. For every write to a logical volume, the LVM generates a write request for every mirror copy. A problem arises if the system crashes in the middle of processing a mirrored write (before all copies are written). If mirror write consistency recovery is requested for a logical volume, the LVM keeps additional information to allow recovery of these inconsistent mirrors. Mirror write consistency recovery should be performed for most mirrored logical volumes. Logical volumes, such as page space, that do not use the existing data when the volume group is re-varied on do not need this protection. Values for the **mirwrt_consist** field are: **LVM_CONSIST** Mirror-write consistency recovery will be done for this logical volume. **LVM_NOCONSIST** Mirror-write consistency recovery will not be done for this logical volume.
open_close	Specifies if the logical volume is opened or closed. Values for this field are: **LVM_QLV_NOTOPEN** The logical volume is closed. **LVM_QLVOPEN** The logical volume is opened by one or more processes.
mirrors	Specifies an array of pointers to partition map lists (physical volume id, logical partition number, physical partition number, and physical partition state for each copy of the logical partitions for the logical volume). The **ppstate** field can be **LVM_PPFREE**, **LVM_PPALLOC**, or **LVM_PPSTALE**. If a logical partition does not contain any copies, its **pv_id**, **lp_num**, and **pp_num** fields will contain zeros.
stripe_exp	Specifies the log base 2 of the logical volume strip size (the strip size multiplied by the number of disks in an array equals the stripe size). For example, 2^20 is 1048576 (that is, 1 MB). Therefore, if the strip size is 1 MB, the **stripe_exp** field is 20. If the logical volume is not striped, the **stripe_exp** field is 0.
stripe_width	Specifies the number of disks that form the striped logical volume. If the logical volume is not striped, the **striping_width** field is 0.

The *PVName* parameter enables the user to query from a volume group descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off.

Note: The data returned is not guaranteed to be the most recent or correct, and it can reflect a back-level descriptor area.

The *PVName* parameter should specify either the full path name of the physical volume that contains the descriptor area to query, or a single file name that must reside in the **/dev** directory (for example, **rhdisk1**). This parameter must be a null-terminated string between 1 and **LVM_NAMESIZ** bytes, including the null byte, and must represent a raw device entry. If a raw or character device is not specified for the *PVName* parameter, the LVM adds an **r** to the file name to have a raw device name. If there is no raw device entry for this name, the LVM returns the **LVM_NOTCHARDEV** error code.

If a *PVName* parameter is specified, only the **minor_num** field of the *LV_ID* parameter need be supplied. The LVM fills in the **vg_id** field and returns it to the user. If the user wishes to query from the LVM's in-memory copy, the *PVName* parameter should be set to null. When using this method of query, the volume group must be varied on, or an error is returned.

Note: As long as the *PVName* parameter is not null, the LVM will attempt a query from a physical volume and *not* from its in-memory copy of data.

In addition to the *PVName* parameter, the caller passes the ID of the logical volume to be queried (*LV_ID* parameter) and the address of a pointer to the **querylv** structure, specified by the *QueryLV* parameter. The LVM separately allocates the space needed for the **querylv** structure and the struct **pp** arrays, and returns the **querylv** structure's address in the pointer variable passed in by the user. The user is responsible for freeing the space by first freeing the struct **pp** pointers in the **mirrors** array and then freeing the **querylv** structure.

Attention: To prevent corruption when there are many pp arrays, the caller of `lvm_querylv` must set *`QueryLV`*`->mirrors k != NULL`.

Parameters

Item	Description
LV_ID	Points to an **lv_id** structure that specifies the logical volume to query.
QueryLV	Contains the address of a pointer to the **querylv** structure.
PVName	Names the physical volume from which to use the volume group descriptor for the query. This parameter can also be null.

Return Values

If the **lvm_querylv** subroutine is successful, it returns a value of 0.

Error Codes

If the **lvm_querylv** subroutine does not complete successfully, it returns one of the following values:

Item	Description
LVM_ALLOCERR	The subroutine could not allocate enough space for the complete buffer.
LVM_INVALID_MIN_NUM	The minor number of the logical volume is not valid.
LVM_INVALID_PARAM	A parameter passed into the routine is not valid.
LVM_INV_DEVENT	The device entry for the physical volume specified by the *Pvname* parameter is not valid and cannot be checked to determine if it is raw.
LVM_NOTCHARDEV	The physical volume name given does not represent a raw or character device.
LVM_OFFLINE	The volume group containing the logical volume to query was offline. If the query originates from the varied-on volume group's current volume group descriptor area, one of the following error codes is returned:
LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	The mapped file could not be read or written.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, one of the following error codes is returned:

Item	Description
LVM_BADBBDIR	The bad-block directory could not be read or written.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_NOPVVGDA	There are no volume group descriptor areas on the physical volume specified.
LVM_NOTVGMEM	The physical volume specified is not a member of a volume group.
LVM_PVDAREAD	An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_VGDA_BB	A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from the specified physical volume.

lvm_querypv Subroutine

Purpose

Queries a physical volume and returns all pertinent information.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>
```

```
int lvm_querypv (VG_ID, PV_ID, QueryPV, PVName)
struct unique_id * VG_ID;
struct unique_id * PV_ID;
struct querypv ** QueryPV;
char * PVName;
```

Description

Note: The **lvm_querypv** subroutine uses the **sysconfig** system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the **lvm_querypv** subroutine.

The **lvm_querypv** subroutine returns information on the physical volume specified by the *PV_ID* parameter.

The **querypv** structure, defined in the **lvm.h** file, contains the following fields:

```
struct querypv {
      long ppsize;
      long pv_state;
      long pp_count;
      long alloc_ppcount;
      long pvnum_vgdas;
      struct pp_map *pp_map;
      char hotspare;
      struct unique_id  pv_id;
```

```
     long freespace;
}
struct pp_map {
     long pp_state;
     struct lv_id lv_id;
     long lp_num;
     long copy;
     struct unique_id fst_alt_vol;
     long fst_alt_part;
     struct unique_id snd_alt_vol;
     long snd_alt_part;
 }
```

Field	Description
ppsize	Specifies the size of the physical partitions, which is the same for all partitions within a volume group. The size in bytes of a physical partition is 2 to the power of ppsize.
pv_state	Contains the current state of the physical volume.
pp_count	Contains the total number of physical partitions on the physical volume.
alloc_ppcount	Contains the number of allocated physical partitions on the physical volume.

Field	Description
pp_map	Points to an array that has entries for each physical partition of the physical volume. Each entry in this array will contain the pp_state that specifies the state of the physical partition (**LVM_PPFREE**, **LVM_PPALLOC**, or **LVM_PPSTALE**) and the lv_id, field, the ID of the logical volume that it is a member of. The pp_map array also contains the physical volume IDs (fst_alt_vol and snd_alt_vol) and the physical partition numbers (fst_alt_part and snd_alt_part) for the first and second alternate copies of the physical partition, and the logical partition number (lp_num) that the physical partition corresponds to. If the physical partition is free (that is, not allocated), *all* of its **pp_map** fields will be zero. **fst_alt_vol** Contains zeros if the logical partition has only one physical copy. **fst_alt_part** Contains zeros if the logical partition has only one physical copy. **snd_alt_vol** Contains zeros if the logical partition has only one or two physical copies. **snd_alt_part** Contains zeros if the logical partition has only one or two physical copies. **copy** Specifies which copy of a logical partition this physical partition is allocated to. This field will contain one of the following values: **LVM_PRIMARY** Primary and only copy of a logical partition **LVM_PRIMOF2** Primary copy of a logical partition with two physical copies **LVM_PRIMOF3** Primary copy of a logical partition with three physical copies **LVM_SCNDOF2** Secondary copy of a logical partition with two physical copies **LVM_SCNDOF3** Secondary copy of a logical partition with three physical copies **LVM_TERTOF3** Tertiary copy of a logical partition with three physical copies.
pvnum_vgdas	Contains the number of volume group descriptor areas (0, 1, or 2) that are on the specified physical volume.
hotspare	Specifies that the physical volume is a hotspare.
pv_id	Specifies the physical volume identifier.
freespace	Specifies the number of physical partitions in the volume group.

The *PVName* parameter enables the user to query from a volume group descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off. The data returned is not guaranteed to be most recent or correct, and it can reflect a back level descriptor area.

The *PVname* parameter should specify either the full path name of the physical volume that contains the descriptor area to query or a single file name that must reside in the **/dev** directory (for example, **rhdisk1**). This field must be a null-terminated string of from 1 to **LVM_NAMESIZ** bytes, including the

null byte, and represent a raw or character device. If a raw or character device is not specified for the *PVName* parameter, the LVM will add an **r** to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM will return the **LVM_NOTCHARDEV** error code. If a *PVName* is specified, the volume group identifier, `VG_ID`, will be returned by the LVM through the *VG_ID* parameter passed in by the user. If the user wishes to query from the LVM in-memory copy, the *PVName* parameter should be set to null. When using this method of query, the volume group must be varied on, or an error will be returned.

Note: As long as the *PVName* is not null, the LVM will attempt a query from a physical volume and *not* from its in-memory copy of data.

In addition to the *PVName* parameter, the caller passes the *VG_ID* parameter, indicating the volume group that contains the physical volume to be queried, the unique ID of the physical volume to be queried, the *PV_ID* parameter, and the address of a pointer of the type *QueryPV*. The LVM will separately allocate enough space for the **querypv** structure and the struct `pp_map` array and return the address of the **querypv** structure in the *QueryPV* pointer passed in. The user is responsible for freeing the space by freeing the struct *pp_map* pointer and then freeing the *QueryPV* pointer.

Parameters

Item	Description
VG_ID	Points to a **unique_id** structure that specifies the volume group of which the physical volume to query is a member.
PV_ID	Points to a **unique_id** structure that specifies the physical volume to query.
QueryPV	Specifies the address of a pointer to a **querypv** structure.
PVName	Names a physical volume from which to use the volume group descriptor area for the query. This parameter can be null.

Return Values

The **lvm_querypv** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_querypv** subroutine fails it returns one of the following error codes:

Item	Description
LVM_ALLOCERR	The routine cannot allocate enough space for a complete buffer.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_INV_DEVENT	The device entry for the physical volume is invalid and cannot be checked to determine if it is raw.
LVM_OFFLINE	The volume group specified is offline and should be online.

If the query originates from the varied-on volume group's current volume group descriptor area, one of the following error codes may be returned:

Item	Description
LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.

Item	Description
LVM_MAPFRDWR	Either the mapped file could not be read, or it could not be written.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, then one of the following error codes may be returned:

Item	Description
LVM_BADBBDIR	The bad-block directory could not be read or written.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_NOPVVGDA	There are no volume group descriptor areas on this physical volume.
LVM_NOTCHARDEV	A device is not a raw or character device.
LVM_NOTVGMEM	The physical volume is not a member of a volume group.
LVM_PVDAREAD	An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_VGDA_BB	A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from the specified physical volume.

lvm_queryvg Subroutine

Purpose

Queries a volume group and returns pertinent information.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>
```

```
int lvm_queryvg ( VG_ID,  QueryVG,  PVName)
struct unique_id *VG_ID;
struct queryvg **QueryVG;
char *PVName;
```

Description

Note: The **lvm_queryvg** subroutine uses the **sysconfig** system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the **lvm_queryvg** subroutine.

The **lvm_queryvg** subroutine returns information on the volume group specified by the *VG_ID* parameter.

The **queryvg** structure, found in the **lvm.h** file, contains the following fields:

```
struct queryvg {
      long maxlvs;
      long ppsize;
      long freespace;
      long num_lvs;
      long num_pvs;
```

```
      long total_vgdas;
      struct lv_array *lvs;
      struct pv_array *pvs;
      short conc_capable;
      short default_mode;
      short conc_status;
      unsigned int maxpvs;
      unsigned int maxpvpps;
      unsigned int maxvgpps;
      int total_pps;
      char vgtype;
      daddr32_t beg_psn;
}
struct pv_array {
    struct unique_id pv_id;
    char   state;
    char   res[3];
    long pvnum_vgdas;
}
struct lv_array {
     struct lv_id   lv_id;
     char  lvname[LVM_NAMESIZ];
     char   state;
     char   res[3];
}
```

Field	Description
conc_capable	Indicates that the volume group was created concurrent mode capable if the value is equal to one.
conc_status	Indicates that the volume group is varied on in concurrent mode.
beg_psn	Specifies the physical sector number of the first physical partition.
default_mode	The behavior of this value is undefined.
freespace	Contains the number of free physical partitions in this volume group.
lvs	Points to an array of unique IDs, names, and states of the logical volumes in the volume group.
maxlvs	Specifies the maximum number of logical volumes allowed in the volume group.
maxpvs	Specifies the maximum number of physical volumes allowed in the volume group.
maxpvpps	Specifies the maximum number of physical partitions allowed for a physical volume in the volume group.
maxvgpps	Specifies the maximum number of physical partitions allowed for the entire volume group.
num_lvs	Indicates the number of logical volumes.
num_pvs	Indicates the number of physical volumes.
ppsize	Specifies the size of all physical partitions in the volume group. The size in bytes of each physical partitions is 2 to the power of the ppsize field.
pvs	Points to an array of unique IDs, states, and the number of volume group descriptor areas for each of the physical volumes in the volume group.
total_pps	Specifies the total number of physical partitions contained in the volume group.
total_vgdas	Specifies the total number of volume group descriptor areas for the entire volume group.

Field	Description
vgtype	Indicates the type of the volume group. If the value of the vgtype field is zero, the volume group is an original volume group. If the value is one, the volume group is a big volume group. If the value is two, the volume group is a scalable volume group.

The *PVName* parameter enables the user to query from a descriptor area on a specific physical volume instead of from the Logical Volume Manager's (LVM) most recent, in-memory copy of the descriptor area. This method should only be used if the volume group is varied off. The data returned is *not guaranteed* to be most recent or correct, and it can reflect a back level descriptor area. The *Pvname* parameter should specify either the full path name of the physical volume that contains the descriptor area to query or a single file name that must reside in the **/dev** directory (for example, **rhdisk1**). The name must represent a raw device. If a raw or character device is not specified for the *PVName* parameter, the Logical Volume Manager will add an r to the file name in order to have a raw device name. If there is no raw device entry for this name, the LVM returns the **LVM_NOTCHARDEV** error code. This field must be a null-terminated string of from 1 to **LVM_NAMESIZ** bytes, including the null byte. If a *PVName* is specified, the LVM will return the *VG_ID* to the user through the *VG_ID* pointer passed in. If the user wishes to query from the LVM in-memory copy, the *PVName* parameter should be set to null. When using this method of query, the volume group must be varied on, or an error will be returned.

Note: As long as the *PVName* parameter is not null, the LVM will attempt a query from a physical volume and *not* its in-memory copy of data.

In addition to the *PVName* parameter, the caller passes the unique ID of the volume group to be queried (*VG_ID*) and the address of a pointer to a **queryvg** structure. The LVM will separately allocate enough space for the **queryvg** structure, as well as the **lv_array** and **pv_array** structures, and return the address of the completed structure in the *QueryVG* parameter passed in by the user. The user is responsible for freeing the space by freeing the lv and pv pointers and then freeing the *QueryVG* pointer.

Parameters

Item	Description
VG_ID	Points to a **unique_id** structure that specifies the volume group to be queried.
QueryVG	Specifies the address of a pointer to the **queryvg** structure.
PVName	Specifies the name of the physical volume that contains the descriptor area to query and must be the name of a raw device.

Return Values

The **lvm_queryvg** subroutine returns a value of zero upon successful completion.

Error Codes

If the **lvm_queryvg** subroutine fails it returns one of the following error codes:

Item	Description
LVM_ALLOCERR	The subroutine cannot allocate enough space for a complete buffer.
LVM_FORCEOFF	The volume group has been forcefully varied off due to a loss of quorum.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_OFFLINE	The volume group is offline and should be online.

If the query originates from the varied-on volume group's current volume group descriptor area, one of the following error codes may be returned:

Item	Description
LVM_DALVOPN	The volume group reserved logical volume could not be opened.
LVM_INV_DEVENT	The device entry for the physical volume specified by the *PVName* parameter is invalid and cannot be checked to determine if it is raw.
LVM_MAPFBSY	The volume group is currently locked because system management on the volume group is being done by another process.
LVM_MAPFOPN	The mapped file, which contains a copy of the volume group descriptor area used for making changes to the volume group, could not be opened.
LVM_MAPFRDWR	Either the mapped file could not be read, or it could not be written.
LVM_NOTCHARDEV	A device is not a raw or character device.

If a physical volume name has been passed, requesting that the query originate from a specific physical volume, one of the following error codes may be returned:

Item	Description
LVM_BADBBDIR	The bad-block directory could not be read or written.
LVM_LVMRECERR	The LVM record, which contains information about the volume group descriptor area, could not be read.
LVM_NOPVVGDA	There are no volume group descriptor areas on this physical volume.
LVM_NOTVGMEM	The physical volume is not a member of a volume group.
LVM_PVDAREAD	An error occurred while trying to read the volume group descriptor area from the specified physical volume.
LVM_PVOPNERR	The physical volume device could not be opened.
LVM_VGDA_BB	A bad block was found in the volume group descriptor area located on the physical volume that was specified for the query. Therefore, a query cannot be done from this physical volume.

lvm_queryvgs Subroutine

Purpose

Queries volume groups and returns information to online volume groups.

Library

Logical Volume Manager Library (**liblvm.a**)

Syntax

```
#include <lvm.h>

int lvm_queryvgs ( QueryVGS,  Kmid)
struct queryvgs **QueryVGS;
mid_t Kmid;
```

Description

Note: The **lvm_queryvgs** subroutine uses the **sysconfig** system call, which requires root user authority, to query and update kernel data structures describing a volume group. You must have root user authority to use the **lvm_queryvgs** subroutine.

The **lvm_queryvgs** subroutine returns the volume group IDs and major numbers for all volume groups in the system that are online.

The caller passes the address of a pointer to a **queryvgs** structure, and the Logical Volume Manager (LVM) allocates enough space for the structure and returns the address of the structure in the pointer passed in by the user. The caller also passes in a *Kmid* parameter, which identifies the entry point of the logical device driver module:

```
struct queryvgs {
       long num_vgs;
       struct {
       long major_num
       struct unique_id vg_id;
       } vgs  [LVM_MAXVGS];
 }
```

Field	Description
num_vgs	Contains the number of online volume groups on the system. The vgs is an array of the volume group IDs and major numbers of all online volume groups in the system.

Parameters

Item	Description
QueryVGS	Points to the **queryvgs** structure.
Kmid	Identifies the address of the entry point of the logical volume device driver module.

Return Values

The **lvm_queryvgs** subroutine returns a value of 0 upon successful completion.

Error Codes

If the **lvm_queryvgs** subroutine fails, it returns one of the following error codes:

Item	Description
LVM_ALLOCERR	The routine cannot allocate enough space for the complete buffer.
LVM_INVALID_PARAM	An invalid parameter was passed into the routine.
LVM_INVCONFIG	An error occurred while attempting to configure this volume group into the kernel. This error will normally result if the module ID is invalid, if the major number given is already in use, or if the volume group device could not be opened.

longname Subroutine

Purpose

Returns the verbose name of a terminal.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
char *longname(void);
```

Description

The **longname** subroutine generates a verbose description for the current terminal. The maximum length of a verbose description is 128 bytes. It is defined only after the call to the **initscr** or **newterm** subroutines.

The area is overwritten by each call to the **newterm** subroutine, so the value should be saved if you plan on using the **longname** subroutine with multiple terminals.

Return Values

Upon successful completion, the **longname** subroutine returns a pointer to the description specified above. Otherwise, it returns a null pointer on error.

m

The following Base Operating System (BOS) runtime services begin with the letter *m*.

malloc, free, realloc, calloc, mallopt, mallinfo, mallinfo_heap, alloca, valloc, or posix_memalign Subroutine

Purpose

Provides a complete set of memory allocation, reallocation, deallocation, and heap management tools.

Libraries

Berkeley Compatibility Library (**libbsd.a**)

Standard C Library (**libc.a**)

Malloc Subsystem APIs

- malloc
- free
- realloc
- calloc
- mallopt
- mallinfo
- mallinfo_heap
- alloca
- valloc
- posix_memalign

malloc

Syntax (malloc)

```
#include <stdlib.h>
```

```
void *malloc (Size)
size_t Size;
```

Description (malloc)

The **malloc** subroutine returns a pointer to a block of memory of at least the number of bytes specified by the *Size* parameter. The block is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by the **malloc** subroutine is overrun.

Parameters (malloc)

Item	Description
Size	Specifies the size, in bytes, of memory to allocate.

Return Values (malloc)

Upon successful completion, the **malloc** subroutine returns a pointer to space suitably aligned for the storage of any type of object. If the size requested is 0, **malloc** returns NULL in normal circumstances. However, if the program was compiled with the defined **_LINUX_SOURCE_COMPAT** macro, **malloc** returns a valid pointer to a space of size 0.

If the request cannot be satisfied for any reason, the **malloc** subroutine returns NULL.

Error Codes (malloc)

Item	Description
ENOMEM	Insufficient storage space is available to service the request.

free

Syntax (free)

```
#include <stdlib.h>
```

void free (*Pointer*)
void * *Pointer*;

Description (free)

The **free** subroutine deallocates a block of memory previously allocated by the **malloc** subsystem. Undefined results occur if the *Pointer* parameter is not an address that has previously been allocated by the **malloc** subsystem, or if the *Pointer* parameter has already been deallocated. If the *Pointer* parameter is NULL, no action occurs.

Parameters (free)

Item	Description
Pointer	Specifies a pointer to space previously allocated by the **malloc** subsystem.

Return Values (free)

The **free** subroutine does not return a value. Upon successful completion with nonzero arguments, the **realloc** subroutine returns a pointer to the (possibly moved) allocated space. If the *Size* parameter is 0 and the *Pointer* parameter is not null, no action occurs.

Error Codes (free)

The **free** subroutine does not set **errno**.

realloc (free)

Syntax (realloc)

```
#include <stdlib.h>
```

void *realloc (*Pointer*, *Size*)
**void ***Pointer;
size_t *Size*;

Description (realloc)

The **realloc** subroutine changes the size of the memory object pointed to by the *Pointer* parameter to the number of bytes specified by the *Size* parameter. The *Pointer* must point to an address returned by a **malloc** subsystem allocation routine, and must not have been previously deallocated. Undefined results occur if *Pointer* does not meet these criteria.

The contents of the memory object remain unchanged up to the lesser of the old and new sizes. If the current memory object cannot be enlarged to satisfy the request, the **realloc** subroutine acquires a new memory object and copies the existing data to the new space. The old memory object is then freed. If no memory object can be acquired to accommodate the request, the object remains unchanged.

If the *Pointer* parameter is null, the **realloc** subroutine is equivalent to a **malloc** subroutine of the same size.

If the *Size* parameter is 0 and the *Pointer* parameter is not null, the **realloc** subroutine is equivalent to a **free** subroutine of the same size.

Parameters (realloc)

Item	Description
Pointer	Specifies a *Pointer* to space previously allocated by the **malloc** subsystem.
Size	Specifies the new size, in bytes, of the memory object.

Return Values (realloc)

Upon successful completion with nonzero arguments, the **realloc** subroutine returns a pointer to the (possibly moved) allocated space. If the *Size* parameter is 0 and the *Pointer* parameter is not null, return behavior is equivalent to that of the **free** subroutine. If the *Pointer* parameter is null and the *Size* parameter is not zero, return behavior is equivalent to that of the **malloc** subroutine.

Error Codes (realloc)

Item	Description
ENOMEM	Insufficient storage space is available to service the request.

calloc

Syntax (calloc)

```
#include <stdlib.h>

void *calloc (NumberOfElements, ElementSize)
size_t NumberOfElements;
size_t ElementSize;
```

Description (calloc)

The **calloc** subroutine allocates space for an array containing the *NumberOfElements* objects. The *ElementSize* parameter specifies the size of each element in bytes. After the array is allocated, all bits are initialized to 0.

The order and contiguity of storage allocated by successive calls to the **calloc** subroutine is unspecified. The pointer returned points to the first (lowest) byte address of the allocated space. The allocated space is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by the **calloc** subroutine is overrun.

Parameters (calloc)

Item	Description
NumberOfElements	Specifies the number of elements in the array.
ElementSize	Specifies the size, in bytes, of each element in the array.

Return Values (calloc)

Upon successful completion, the **calloc** subroutine returns a pointer to the allocated, zero-initialized array. If the size requested is 0, the **calloc** subroutine returns NULL in normal circumstances. However, if the program was compiled with the macro **_LINUX_SOURCE_COMPAT** defined, the **calloc** subroutine returns a valid pointer to a space of size 0.

If the request cannot be satisfied for any reason, the **calloc** subroutine returns NULL.

Error Codes (calloc)

Item	Description
ENOMEM	Insufficient storage space is available to service the request.

mallopt

Syntax (mallopt)

```
#include <malloc.h>
#include <stdlib.h>
```

```
int mallopt (Command, Value)
int Command;
int Value;
```

Description (mallopt)

The **mallopt** subroutine is provided for source-level compatibility with the System V **malloc** subroutine. The **mallopt** subroutine supports the following commands:

Table 20. Commands and effects

Command	Value	Effect
M_MXFAST	0	If called before any other **malloc** subsystem subroutine, this enables the Default allocation policy for the process.
M_MXFAST	1	If called before any other **malloc** subsystem subroutine, this enables the 3.1 allocation policy for the process.
M_DISCLAIM	0	If called while the Default Allocator is enabled, all free memory in the process heap is disclaimed.

Table 20. Commands and effects (continued)		
Command	**Value**	**Effect**
M_MALIGN	N	If called at runtime, sets the default `malloc` allocation alignment to the value *N*. The *N* value must be a power of 2 (greater than or equal to the size of a pointer).

Parameters (mallopt)

Item	Description
Command	Specifies the **mallopt** command to be executed.
Value	Specifies the size of each element in the array.

Return Values (mallopt)

Upon successful completion, the **mallopt** subroutine returns 0. Otherwise, 1 is returned. If an invalid alignment is requested (one that is not a power of 2), `mallopt` fails with a return value of 1, although subsequent calls to `malloc` are unaffected and continue to provide the alignment value from before the failed `mallopt` call.

Error Codes (mallopt)

The **mallopt** subroutine does not set **errno**.

mallinfo

Syntax (mallinfo)

```
#include <malloc.h>
#include <stdlib.h>

struct mallinfo mallinfo();
```

Description (mallinfo)

The **mallinfo** subroutine can be used to obtain information about the heap managed by the **malloc** subsystem.

Return Values (mallinfo)

The **mallinfo** subroutine returns a structure of type **struct mallinfo**, filled in with relevant information and statistics about the heap. The contents of this structure can be interpreted using the definition of **struct mallinfo** in the **/usr/include/malloc.h** file.

Error Codes (mallinfo)

The **mallinfo** subroutine does not set **errno**.

mallinfo_heap

Syntax (mallinfo_heap)

```
#include <malloc.h>
#include <stdlib.h>
```

struct mallinfo_heap mallinfo_heap (*Heap*)
int *Heap*;

Description (mallinfo_heap)

In a multiheap context, the **mallinfo_heap** subroutine can be used to obtain information about a specific heap managed by the **malloc** subsystem.

Parameters (mallinfo_heap)

Item	Description
Heap	Specifies which heap to query.

Return Values (mallinfo_heap)

mallinfo_heap returns a structure of type **struct mallinfo_heap**, filled in with relevant information and statistics about the heap. The contents of this structure can be interpreted using the definition of **struct mallinfo_heap** in the **/usr/include/malloc.h** file.

Error Codes (mallinfo_heap)

The **mallinfo_heap** subroutine does not set **errno**.

alloca

Syntax (alloca)

```
#include <stdlib.h>
```

char *alloca (*Size*)
int *Size*;

Description (alloca)

The **alloca** subroutine returns a pointer to a block of memory of at least the number of bytes specified by the *Size* parameter. The space is allocated from the stack frame of the caller and is automatically freed when the calling subroutine returns.

If the **alloca** subroutine is used in code compiled with the IBM XL C for AIX compiler, `#pragma alloca` must be added to the source code before referencing the **alloca** subroutine. Alternatively, you can add the **-ma** compiler flag or the **<alloca.h>** header file.

Parameters (alloca)

Item	Description
Size	Specifies the size, in bytes, of memory to allocate.

Return Values (alloca)

The **alloca** subroutine returns a pointer to space of the requested size.

Error Codes (alloca)

The **alloca** subroutine does not set **errno**.

valloc

Syntax (valloc)

```
#include <stdlib.h>
```

void *valloc (*Size*)
size_t *Size*;

Description (valloc)

The **valloc** subroutine is supported as a compatibility interface in the Berkeley Compatibility Library (**libbsd.a**), as well as in **libc.a**. The **valloc** subroutine has the same effect as **malloc**, except that the allocated memory is aligned to a multiple of the value returned by **sysconf** (**_ SC_PAGESIZE**).

Parameters (valloc)

Item	Description
Size	Specifies the size, in bytes, of memory to allocate.

Return Values (valloc)

Upon successful completion, the **valloc** subroutine returns a pointer to a memory object that is *Size* bytes in length, aligned to a page-boundary. Undefined results occur if the space assigned by the **valloc** subroutine is overrun.

If the request cannot be satisfied for any reason, **valloc** returns NULL.

Error Codes (valloc)

Item	Description
ENOMEM	Insufficient storage space is available to service the request.

posix_memalign

Syntax (posix_memalign)

```
#include <stdlib.h>
```

int posix_memalign(void **Pointer2Pointer, *Align*, *Size*)
**void ** *Pointer2Pointer*;
size_t *Align*;
size_t *Size*;

Description (posix_memalign)

The **posix_memalign** subroutine allocates *Size* bytes of memory aligned on a boundary specified by *Align*. The address of this memory is stored in *Pointer2Pointer*.

Parameters (posix_memalign)

Item	Description
Pointer2Pointer	Specifies the location in which the address should be copied.
Align	Specifies the alignment of the allocated memory, in bytes. The *Align* parameter must be a power-of-two multiple of the size of a pointer.
Size	Specifies the size, in bytes, of memory to allocate.

Return Values (posix_memalign)

Upon successful completion, **posix_memalign** returns 0. Otherwise, an error number is returned to indicate the error.

Error Codes (posix_memalign)

Item	Description
EINVAL	The value of *Align* is not a power-of-two multiple of the size of a pointer.
ENOMEM	Insufficient storage space is available to service the request.

madd, msub, mult, mdiv, pow, gcd, invert, rpow, msqrt, mcmp, move, min, omin, fmin, m_in, mout, omout, fmout, m_out, sdiv, or itom Subroutine

Purpose

Multiple-precision integer arithmetic.

Library

Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <mp.h>
#include <stdio.h>
```

typedef struct mint {int *Length*; **short *** *Value*} **MINT;**

madd(*a*, *b*, *c*)
msub(*a*,*b*,*c*)
mult(*a*,*b*,*c*)
mdiv(*a*,*b*, *q*, *r*)
pow(*a*,*b*, *m*,*c*)
gcd(*a*,*b*,*c*)
invert(*a*,*b*,*c*)
rpow(*a*,*n*,*c*)
msqrt(*a*,*b*,*r*)
mcmp(*a*,*b*)
move(*a*,*b*)
min(*a*)
omin(*a*)
fmin(*a*,*f*)
m_in(*a*, *n*,*f*)

```
mout(a)
omout(a)
fmout(a,f)
m_out(a,n,f)
MINT *a, *b, *c, *m, *q, *r;
FILE * f;
int n;
```

```
sdiv(a,n,q,r)
MINT *a, *q;
short n;
short *r;
```

```
MINT *itom(n)
```

Description

These subroutines perform arithmetic on integers of arbitrary *Length*. The integers are stored using the defined type **MINT**. Pointers to a **MINT** can be initialized using the **itom** subroutine, which sets the initial *Value* to *n*. After that, space is managed automatically by the subroutines.

The **madd** subroutine, **msub** subroutine, and **mult** subroutine assign to *c* the sum, difference, and product, respectively, of *a* and *b*.

The **mdiv** subroutine assigns to *q* and *r* the quotient and remainder obtained from dividing *a* by *b*.

The **sdiv** subroutine is like the **mdiv** subroutine except that the divisor is a short integer *n* and the remainder is placed in a short whose address is given as *r*.

The **msqrt** subroutine produces the integer square root of *a* in *b* and places the remainder in *r*.

The **rpow** subroutine calculates in c the value of *a* raised to the (regular integral) power *n*, while the **pow** subroutine calculates this with a full multiple precision exponent *b* and the result is reduced modulo *m*.

Note: The **pow** subroutine is also present in the IEEE Math Library, **libm.a**, and the System V Math Library, **libmsaa.a**. The **pow** subroutine in **libm.a** or **libmsaa.a** may be loaded in error unless the **libbsd.a** library is listed before the **libm.a** or **libmsaa.a** library on the command line.

The **gcd** subroutine returns the greatest common denominator of *a* and *b* in *c*, and the **invert** subroutine computes *c* such that *a***c* mod *b*=1, for *a* and *b* relatively prime.

The **mcmp** subroutine returns a negative, 0, or positive integer value when *a* is less than, equal to, or greater than *b*, respectively.

The **move** subroutine copies *a* to *b*. The **min** subroutine and **mout** subroutine do decimal input and output while the **omin** subroutine and **omout** subroutine do octal input and output. More generally, the **fmin** subroutine and **fmout** subroutine do decimal input and output using file *f*, and the **m_in** subroutine and **m_out** subroutine do inputs and outputs with arbitrary radix *n*. On input, records should have the form of strings of digits terminated by a new line; output records have a similar form.

Programs that use the multiple-precision arithmetic functions must link with the **libbsd.a** library.

Bases for input and output should be less than or equal to 10.

pow is also the name of a standard math library routine.

Parameters

Item	Description
Length	Specifies the length of an integer.
Value	Specifies the initial value to be used in the routine.
a	Specifies the first operand of the multiple-precision routines.

Item	Description
b	Specifies the second operand of the multiple-precision routines.
c	Contains the integer result.
f	A pointer of the type **FILE** that points to input and output files used with input/output routines.
m	Indicates modulo.
n	Provides a value used to specify radix with **m_in** and **m_out**, power with **rpow**, and divisor with **sdiv**.
q	Contains the quotient obtained from **mdiv**.
r	Contains the remainder obtained from **mdiv, sdiv,** and **msqrt**.

Error Codes

Error messages and core images are displayed as a result of illegal operations and running out of memory.

Files

Item	Description
/usr/lib/libbsd.a	Object code library.

madvise Subroutine

Purpose

Advises the system of expected paging behavior.

Library

Standard C Library **(libc.a)**.

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int madvise( addr,  len,  behav)
caddr_t addr;
size_t len;
int behav;
```

Description

The **madvise** subroutine permits a process to advise the system about its expected future behavior in referencing a mapped file region or anonymous memory region.

The **madvise** subroutine has no functionality and is supported for compatibility only.

Parameters

Item	Description
addr	Specifies the starting address of the memory region. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

Item	Description
len	Specifies the length, in bytes, of the memory region. If the *len* value is not a multiple of page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region will be rounded up to the next multiple of the page size.
behav	Specifies the future behavior of the memory region. The following values for the *behav* parameter are defined in the **/usr/include/sys/mman.h** file: **Value** **Paging Behavior Message** **MADV_NORMAL** The system provides no further special treatment for the memory region. **MADV_RANDOM** The system expects random page references to that memory region. **MADV_SEQUENTIAL** The system expects sequential page references to that memory region. **MADV_WILLNEED** The system expects the process will need these pages. **MADV_DONTNEED** The system expects the process does not need these pages. **MADV_SPACEAVAIL** The system will ensure that memory resources are reserved.

Return Values

When successful, the **madvise** subroutine returns 0. Otherwise, it returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **madvise** subroutine is unsuccessful, the **errno** global variable can be set to one of the following values:

Item	Description
EINVAL	The *behav* parameter is invalid.
ENOSPC	C:\A Workspace\71S\src\idd\en_US\basetrf1The *behav* parameter specifies **MADV_SPACEAVAIL** and resources cannot be reserved.

makecontext or swapcontext Subroutine

Purpose

Modifies the context specified by *ucp*.

Library

(**libc.a**)

Syntax

#include <ucontext.h>

void makecontext (ucontext_t ****ucp*, **(void ****func*) **(), **int** *argc*, ...**); int swapcontext (uncontext_t ****oucp*, **const uncontext_t ****ucp*);

Description

The **makecontext** subroutine modifies the context specified by *ucp*, which has been initialized using **getcontext** subroutine. When this context is resumed using **swapcontext** subroutine or **setcontext** subroutine, program execution continues by calling *func* parameter, passing it the arguments that follow *argc* in the **makecontext** subroutine.

Before a call is made to **makecontext** subroutine, the context being modified should have a stack allocated for it. The value of *argc* must match the number of integer argument passed to *func* parameter, otherwise the behavior is undefined.

The **uc_link** member is used to determine the context that will be resumed when the context being modified by **makecontext** subroutine returns. The **uc_link** member should be initialized prior to the call to **makecontext** subroutine.

The **swapcontext** subroutine function saves the current context in the context structure pointed to by *oucp* parameter and sets the context to the context structure pointed to by *ucp*.

Parameters

Item	Description
ucp	A pointer to a user structure.
oucp	A pointer to a user structure.
func	A pointer to a function to be called when *ucp* is restored.
argc	The number of arguments being passed to *func* parameter.

Return Values

On successful completion, **swapcontext** subroutine returns 0. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Item	Description
-1	Not successful and the **errno** global variable is set to one of the following error codes.

Error Codes

Item	Description
ENOMEM	The *ucp* argument does not have enough stack left to complete the operation.

makenew Subroutine

Purpose

Creates a new window buffer and returns a pointer.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
WINDOW *makenew( )
```

Description

The **makenew** subroutine creates a new window buffer and returns a pointer to it. The **makenew** subroutine is called by the **newwin** subroutine to create the window structure. The **makenew** subroutine should not be called directly by a program.

matherr Subroutine

Purpose

Math error handling function.

Library

System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>
```

```
int matherr (x)
struct exception *x;
```

Description

The **matherr** subroutine is called by math library routines when errors are detected.

You can use **matherr** or define your own procedure for handling errors by creating a function named `matherr` in your program. Such a user-designed function must follow the same syntax as **matherr.** When an error occurs, a pointer to the exception structure will be passed to the user-supplied `matherr` function. This structure, which is defined in the **math.h** file, includes:

```
int type;
char *name;
double arg1, arg2, retval;
```

Parameters

Item	Description
type	Specifies an integer describing the type of error that has occurred from the following list defined by the **math.h** file: **DOMAIN** Argument domain error **SING** Argument singularity **OVERFLOW** Overflow range error **UNDERFLOW** Underflow range error **TLOSS** Total loss of significance **PLOSS** Partial loss of significance.
name	Points to a string containing the name of the routine that caused the error.
arg1	Points to the first argument with which the routine was invoked.
arg2	Points to the second argument with which the routine was invoked.
retval	Specifies the default value that is returned by the routine unless the user's **matherr** function sets it to a different value.

Return Values

If the user's **matherr** function returns a non-zero value, no error message is printed, and the **errno** global variable will not be set.

Error Codes

If the function **matherr** is not supplied by the user, the default error-handling procedures, described with the math library routines involved, will be invoked upon error. In every case, the **errno** global variable is set to **EDOM** or **ERANGE** and the program continues.

MatchAllAuths, MatchAnyAuths, MatchAllAuthsList, or MatchAnyAuthsList Subroutine

Purpose

Compare authorizations.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int MatchAllAuths(CommaListOfAuths)
char *CommaListOfAuths;
```

```
int MatchAllAuthsList(CommaListOfAuths, NSListOfAuths)
char *CommaListOfAuths;
char *NSListOfAuths;
```

```
int MatchAnyAuths(CommaListOfAuths)
char *CommaListOfAuths;
```

```
int MatchAnyAuthsList(CommaListOfAuths, NSListOfAuths)
char *CommaListOfAuths;
char *NSListOfAuths;
```

Description

The **MatchAllAuthsList** subroutine compares the *CommaListOfAuths* against the *NSListOfAuths*. It returns a non-zero value if all the authorizations in *CommaListOfAuths* are contained in *NSListOfAuths*. The **MatchAllAuths** subroutine calls the **MatchAllAuthsList** subroutine passing in the results of the **GetUserAuths** subroutine in place of *NSListOfAuths*. If *NSListOfAuths* contains the OFF keyword, **MatchAllAuthsList** will return a zero value. If *NSListOfAuths* contains the ALL keyword and not the OFF keyword, **MatchAllAuthsList** will return a non-zero value.

The **MatchAnyAuthsList** subroutine compares the *CommaListOfAuths* against the *NSListOfAuths*. It returns a non-zero value if one or more of the authorizations in *CommaListOfAuths* are contained in *NSListOfAuths*. The **MatchAnyAuths** subroutine calls the **MatchAnyAuthsList** subroutine passing in the results of the **GetUserAuths** subroutine in place of *NSListOfAuths*. If *NSListOfAuths* contains the OFF keyword, **MatchAnyAuthsList** will return a zero value. If *NSListOfAuths* contains the ALL keyword and not the OFF keyword, **MatchAnyAuthsList** will return a non-zero value.

Parameters

Item	Description
CommaListOfAuths	Specifies one or more authorizations, each separated by a comma.
NSListOfAuths	Specifies zero or more authorizations. Each authorization is null terminated. The last entry in the list must be a null string.

Return Values

The subroutines return a non-zero value if a proper match was found. Otherwise, they will return zero. If an error occurs, the subroutines will return zero and set **errno** to indicate the error. If the subroutine returns zero and no error occurred, **errno** is set to zero.

maxlen_sl, maxlen_cl, and maxlen_tl Subroutines

Purpose

Determine the maximum size of the sensitivity label (SL), the clearance label (CL), and the integrity label (TL).

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>

int maxlen_sl (void)

int maxlen_cl (void)
```

```
int maxlen_tl (void)
```

Description

The **maxlen_sl** subroutine retrieves the maximum possible length of a sensitivity label (SL) that is defined in the current label encodings file.

The **maxlen_cl** subroutine retrieves the maximum possible length of a clearance label (CL) that is defined in the current label encodings file.

The **maxlen_tl** subroutine retrieves the maximum possible length of a integrity label (TL) that is defined in the current label encodings file.

For a label encoding file, the maximum length of a SL, a CL, or a TL is calculated and is constant, unless the labels configuration is modified.

Requirement: Must initialize the database before running these subroutines.

Files Access

Mode	File
r	**/etc/security/enc/LabelEncodings**

Return Values

If successful, these subroutines return the maximum length of NULL terminated label. Otherwise, they return a value of -1.

Error Codes

If these subroutines fail, they set one of the following error codes:

Item	Description
ENOTREADY	The database is not initialized.

mblen Subroutine

Purpose

Determines the length in bytes of a multibyte character.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

int mblen( MbString,  Number)
const char *MbString;
size_t Number;
```

Description

The **mblen** subroutine determines the length, in bytes, of a multibyte character.

Parameters

Item	Description
Mbstring	Points to a multibyte character string.
Number	Specifies the maximum number of bytes to consider.

Return Values

The **mblen** subroutine returns 0 if the *MbString* parameter points to a null character. It returns -1 if a character cannot be formed from the number of bytes specified by the *Number* parameter. If *MbString* is a null pointer, 0 is returned.

mbrlen Subroutine

Purpose

Get number of bytes in a character (restartable).

Library

Standard Library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
size_t mbrlen (const char *s, size_t n, mbstate_t *ps )
```

Description

If *s* is not a null pointer, **mbrlen** determines the number of bytes constituting the character pointed to by *s*. It is equivalent to:

```
mbstate_t internal;
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal);
```

If *ps* is a null pointer, the **mbrlen** function uses its own internal **mbstate_t** object, which is initialized at program startup to the initial conversion state. Otherwise, the mbstate_t object pointed to by *ps* is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls **mbrlen**.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

The **mbrlen** function returns the first of the following that applies:

Item	Description
0	If the next **n** or fewer bytes complete the character that corresponds to the null wide-character
positive	If the next **n** or fewer bytes complete a valid character; the value returned is the number of bytes that complete the character.

Item	Description
(size_t)-2	If the next **n** bytes contribute to an incomplete but potentially valid character, and all n bytes have been processed. When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
(size_t)-1	If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid character. In this case, EILSEQ is stored in **errno** and the conversion state is undefined.

Error Codes

The **mbrlen** function may fail if:

Item	Description
EINVAL	**ps** points to an object that contains an invalid conversion state.
EILSEQ	Invalid character sequence is detected.

mbrtoc16, mbrtoc32 Subroutine

Purpose

The **mbrtoc16** and **mbrtoc32** subroutine converts a 16-bit wide character (UTF-16) and a 32-bit wide character (UTF-32) to the corresponding multibyte character of the current locale.

Library

Standard C library (**libc.a**)

Syntax

```
#include <uchar.h>
size_t mbrtoc16 (char16_t * restrict pc16,  const char * restrict s, size_t n, mbstate_t *
restrict ps);

size_t mbrtoc32 (char32_t * restrict pc32,  const char * restrict s, size_t n, mbstate_t *
restrict ps);
```

Description

If the value of the **s** parameter is a null pointer, the **mbrtoc16** subroutine is equivalent to the following call:

```
mbrtoc16(NULL, "", 1, ps).
```

In this case, the values of the **pc16** and **n** parameters are ignored.

If the value of the **s** parameter is not a null pointer, the **mbrtoc16** subroutine inspects the value of **n** bytes beginning with the byte specified by the **s** parameter to determine the number of bytes that is needed to complete the next multibyte character, including any shift sequences.

If the subroutine determines that the next multibyte character is complete and valid, the subroutine determines the values of the corresponding wide characters. If the value of the **pc16** subroutine is not a null pointer, the subroutine stores the value of the first character in the object specified by the **pc16** parameter.

If the value of the **s** parameter is a null pointer, the **mbrtoc32** subroutine is equivalent to the following call:

```
mbrtoc32(NULL, "", 1, ps).
```

In this case, the values of the **pc32** and **n** parameters are ignored.

If the value of the **s** parameter is not a null pointer, the **mbrtoc32** subroutine inspects the greater value of **n** bytes beginning with the byte specified by the **s** parameter to determine the number of bytes that is needed to complete the next multibyte character, including any shift sequences.

If the subroutine determines that the next multibyte character is complete and valid, the subroutine determines the values of the corresponding wide characters. If the value of the **pc32** subroutine is not a null pointer, the subroutine stores the value of the first character in the object specified by the **pc32** parameter.

Subsequent calls will store the successive wide characters without using any additional input until all the characters are stored. If the corresponding wide character is a null wide character, the resulting state that is described is the initial conversion state.

Note: The **mbrtoc16** and **mbrtoc32** subroutines includes the **ps** parameter which is of the type pointer to **mbstate_t** that points to an object which describes the current conversion state of the associated multibyte character sequence, which the subroutines alter as necessary. If **ps** is a null pointer, each subroutine uses its own internal **mbstate_t** object. The **mbrtoc16** and **mbrtoc32** subroutines do not avoid data races with other calls to the same subroutine.

Parameters

Item	Description
n	Specifies the number of bytes to be examined to determine the next multibyte character.
pc16, pc32	Specifies the location of the first wide character to be stored.
ps	Specifies the state of the conversion.
s	Specifies the beginning of the bytes that are examined.

Example

- The **mbstate_t** pointer can be used as follows:

```
 mbstate_t ss = 0;
```

```
int x = mbrtoc16(&c16, mbs, MB_CUR_MAX, &ss);
```

Return Values

The **mbrtoc16** and **mbrtoc32** subroutine returns any one of the following values that applies to the current conversion state.

Item	Description
0	If the next **n** or fewer bytes complete the multibyte character that corresponds to the null wide-character (which is the value that is stored) from 1 to **n** and if the next **n** or fewer bytes complete a valid multibyte character (which is the value that is stored), the value that is returned is the number of bytes that complete the multibyte character.
(size_t)(-3)	If the next character resulting from a previous call has been stored, no bytes from the input is used by this call.

Item	Description
(size_t)(-2)	If the next **n** bytes contribute to an incomplete but a valid multibyte character, and all **n** bytes are processed, and no value is stored.
(size_t)(-1)	If an encoding error occurs, that is the next **n** or fewer bytes do not contribute to a complete and valid multibyte character (no value is stored), the value of the **EILSEQ** macro is stored in the **errno** variable. The conversion state is unspecified.

Error codes

The **mbrtoc16** and **mbrtoc32** subroutine are unsuccessful if the following error code is set.

Item	Description
EILSEQ	Indicates an invalid multibyte character sequence.

Files

The **uchar.h** file defines standard macros, data types, and subroutines.

mbrtowc Subroutine

Purpose

Convert a character to a wide-character code (restartable).

Library

Standard Library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
size_t mbrtowc (wchar_t * pwc, const char * s, size_t n, mbstate_t * ps) ;
```

Description

If *s* is a null pointer, the **mbrtowc** function is equivalent to the call:

```
mbrtowc(NULL, '''', 1, ps)
```

In this case, the values of the arguments **pwc** and **n** are ignored.

If *s* is not a null pointer, the **mbrtowc** function inspects at most *n* bytes beginning at the byte pointed to by *s* to determine the number of bytes needed to complete the next character (including any shift sequences). If the function determines that the next character is completed, it determines the value of the corresponding wide-character and then, if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc.* If the corresponding wide-character is the null wide-character, the resulting state described is the initial conversion state.

If *ps* is a null pointer, the **mbrtowc** function uses its own internal **mbstate_t** object, which is initialized at program startup to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls **mbrtowc**.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

The **mbrtowc** function returns the first of the following that applies:

Item	Description
0	If the next n or fewer bytes complete the character that corresponds to the null wide-character (which is the value stored).
positive	If the next n or fewer bytes complete a valid character (which is the value stored); the value returned is the number of bytes that complete the character.
(size_t)-2	If the next n bytes contribute to an incomplete but potentially valid character, and all n bytes have been processed (no value is stored). When n has at least the value of the MB_CUR_MAX macro, this case can only occur if s points at a sequence of redundant shift sequences (for implementations with state-dependent encodings).
(size_t)-1	If an encoding error occurs, in which case the next n or fewer bytes do not contribute to a complete and valid character (no value is stored). In this case, EILSEQ is stored in errno and the conversion state is undefined.

Error Codes

The **mbrtowc** function may fail if:

Item	Description
EINVAL	ps points to an object that contains an invalid conversion state.
EILSEQ	Invalid character sequence is detected.

mbsadvance Subroutine

Purpose

Advances to the next multibyte character.

Note: The **mbsadvance** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>
```

```
char *mbsadvance ( S)
const char *S;
```

Description

The **mbsadvance** subroutine locates the next character in a multibyte character string. The **LC_CTYPE** category affects the behavior of the **mbsadvance** subroutine.

Parameters

Item	Description
S	Contains a multibyte character string.

Return Values

If the *S* parameter is not a null pointer, the **mbsadvance** subroutine returns a pointer to the next multibyte character in the string pointed to by the *S* parameter. The character at the head of the string pointed to by the *S* parameter is skipped. If the *S* parameter is a null pointer or points to a null string, a null pointer is returned.

Examples

To find the next character in a multibyte string, use the following:

```
#include <mbstr.h>
#include <locale.h>
#include <stdlib.h>
```

```
main()
{
   char *mbs, *pmbs;
   (void) setlocale(LC_ALL, "");
   /*
   ** Let mbs point to the beginning of a multi-byte string.
   */
   pmbs = mbs;
   while(pmbs){
      pmbs = mbsadvance(mbs);
      /* pmbs points to the next multi-byte character
      ** in mbs */
}
```

mbscat, mbscmp, or mbscpy Subroutine

Purpose

Performs operations on multibyte character strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>
```

```
char *mbscat(MbString1, MbString2)
char *MbString1, *MbString2;
```

```
int mbscmp(MbString1, MbString2)
char *MbString1, *MbString2;
```

```
char *mbscpy(MbString1, MbString2)
char *MbString1, *MbString2;
```

Description

The **mbscat**, **mbscmp**, and **mbscpy** subroutines operate on null-terminated multibyte character strings.

The **mbscat** subroutine appends multibyte characters from the *MbString2* parameter to the end of the *MbString1* parameter, appends a null character to the result, and returns *MbString1*.

The **mbscmp** subroutine compares multibyte characters based on their collation weights as specified in the **LC_COLLATE** category. The **mbscmp** subroutine compares the *MbString1* parameter to the *MbString2* parameter, and returns an integer greater than 0 if *MbString1* is greater than *MbString2*. It returns 0 if the strings are equivalent and returns an integer less than 0 if *MbString1* is less than *MbString2*.

The **mbscpy** subroutine copies multibyte characters from the *MbString2* parameter to the *MbString1* parameter and returns *MbString1*. The copy operation terminates with the copying of a null character.

mbschr Subroutine

Purpose

Locates a character in a multibyte character string.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>

char *mbschr( MbString,  MbCharacter)
char *MbString;
mbchar_t MbCharacter;
```

Description

The **mbschr** subroutine locates the first occurrence of the value specified by the *MbCharacter* parameter in the string pointed to by the *MbString* parameter. The *MbCharacter* parameter specifies a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

The **LC_CTYPE** category affects the behavior of the **mbschr** subroutine.

Parameters

Item	Description
MbString	Points to a multibyte character string.
MbCharacter	Specifies a multibyte character represented as an integer.

Return Values

The **mbschr** subroutine returns a pointer to the value specified by the *MbCharacter* parameter within the multibyte character string, or a null pointer if that value does not occur in the string.

mbsinit Subroutine

Purpose

Determine conversion object status.

Library

Standard Library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
int mbsinit (const mbstate_t * ps) ;
```

Description

If *ps* is not a null pointer, the **mbsinit** function determines whether the object pointed to by ps describes an initial conversion state.

The **mbstate_t** object is used to describe the current conversion state from a particular character sequence to a wide-character sequence (or vice versa) under the rules of a particular setting of the LC_CTYPE category of the current locale.

The initial conversion state corresponds, for a conversion in either direction, to the beginning of a new character sequence in the initial shift state. A zero valued **mbstate_t** object is at least one way to describe an initial conversion state. A zero valued **mbstate_t** object can be used to initiate conversion involving any character sequence, in any LC_CTYPE category setting.

Return Values

The **mbsinit** function returns non-zero if **ps** is a null pointer, or if the pointed-to object describes an initial conversion state; otherwise, it returns zero.

If an **mbstate_t** object is altered by any of the functions described as `restartable`, and is then used with a different character sequence, or in the other conversion direction, or with a different LC_CTYPE category setting than on earlier function calls, the behavior is undefined.

mbsinvalid Subroutine

Purpose

Validates characters of multibyte character strings.

Note: The **mbsinvalid** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>
```

```
char *mbsinvalid ( S)
const char *S;
```

Description

The **mbsinvalid** subroutine examines the string pointed to by the *S* parameter to determine the validity of characters. The **LC_CTYPE** category affects the behavior of the **mbsinvalid** subroutine.

Parameters

Item	Description
S	Contains a multibyte character string.

Return Values

The **mbsinvalid** subroutine returns a pointer to the byte following the last valid multibyte character in the *S* parameter. If all characters in the *S* parameter are valid multibyte characters, a null pointer is returned. If the *S* parameter is a null pointer, the behavior of the **mbsinvalid** subroutine is unspecified.

mbslen Subroutine

Purpose

Determines the number of characters (code points) in a multibyte character string.

Note: The **mbslen** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

size_t mbslen( MbString)
char *mbs;
```

Description

The **mbslen** subroutine determines the number of characters (code points) in a multibyte character string. The **LC_CTYPE** category affects the behavior of the **mbslen** subroutine.

Parameters

Item	Description
MbString	Points to a multibyte character string.

Return Values

The **mbslen** subroutine returns the number of multibyte characters in a multibyte character string. It returns 0 if the *MbString* parameter points to a null character or if a character cannot be formed from the string pointed to by this parameter.

mbsncat, mbsncmp, or mbsncpy Subroutine

Purpose

Performs operations on a specified number of null-terminated multibyte characters.

Note: These subroutines are specific to the manufacturer. They are not defined in the POSIX, ANSI, or X/Open standards. Use of these subroutines may affect portability.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>
```

```
char *mbsncat(MbString1, MbString2, Number)
char * MbString1, * MbString2;
size_t  Number;
```

```
int mbsncmp(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size_t Number;
```

```
char *mbsncpy(MbString1, MbString2, Number)
char *MbString1, *MbString2;
size_t Number;
```

Description

The **mbsncat**, **mbsncmp**, and **mbsncpy** subroutines operate on null-terminated multibyte character strings.

The **mbsncat** subroutine appends up to the specified maximum number of multibyte characters from the *MbString2* parameter to the end of the *MbString1* parameter, appends a null character to the result, and then returns the *MbString1* parameter.

The **mbsncmp** subroutine compares the collation weights of multibyte characters. The **LC_COLLATE** category specifies the collation weights for all characters in a locale. The **mbsncmp** subroutine compares up to the specified maximum number of multibyte characters from the *MbString1* parameter to the *MbString2* parameter. It then returns an integer greater than 0 if *MbString1* is greater than *MbString2*. It returns 0 if the strings are equivalent. It returns an integer less than 0 if *MbString1* is less than *MbString2*.

The **mbsncpy** subroutine copies up to the value of the *Number* parameter of multibyte characters from the *MbString2* parameter to the *MbString1* parameter and returns *MbString1*. If *MbString2* is shorter than *Number* multi-byte characters, *MbString1* is padded out to *Number* characters with null characters.

Parameters

Item	Description
MbString1	Contains a multibyte character string.
MbString2	Contains a multibyte character string.
Number	Specifies a maximum number of characters.

mbspbrk Subroutine

Purpose

Locates the first occurrence of multibyte characters or code points in a string.

Note: The **mbspbrk** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>
```

char *mbspbrk(*MbString1*, *MbString2***)**
**char ***MbString1*, ***MbString2*;

Description

The **mbspbrk** subroutine locates the first occurrence in the string pointed to by the *MbString1* parameter, of any character of the string pointed to by the *MbString2* parameter.

Parameters

Item	Description
MbString1	Points to the string being searched.
MbString2	Pointer to a set of characters in a string.

Return Values

The **mbspbrk** subroutine returns a pointer to the character. Otherwise, it returns a null character if no character from the string pointed to by the *MbString2* parameter occurs in the string pointed to by the *MbString1* parameter.

mbsrchr Subroutine

Purpose

Locates a character or code point in a multibyte character string.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>
```

char *mbsrchr(*MbString*, *MbCharacter***)**
**char ***MbString*;
int *MbCharacter*;

Description

The **mbschr** subroutine locates the last occurrence of the *MbCharacter* parameter in the string pointed to by the *MbString* parameter. The *MbCharacter* parameter is a multibyte character represented as an integer. The terminating null character is considered to be part of the string.

Parameters

Item	Description
MbString	Points to a multibyte character string.
MbCharacter	Specifies a multibyte character represented as an integer.

Return Values

The **mbsrchr** subroutine returns a pointer to the *MbCharacter* parameter within the multibyte character string. It returns a null pointer if *MbCharacter* does not occur in the string.

mbsrtowcs Subroutine

Purpose

Convert a character string to a wide-character string (restartable).

Library

Standard Library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
size_t mbsrtowcs ((wchar_t * dst, const char ** src, size_t len, mbstate_t * ps) ;
```

Description

The **mbsrtowcs** function converts a sequence of characters, beginning in the conversion state described by the object pointed to by *ps*, from the array indirectly pointed to by **src** into a sequence of corresponding wide-characters. If *dst* is not a null pointer, the converted characters are stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops early in either of the following cases:

- When a sequence of bytes is encountered that does not form a valid character.
- When *len* codes have been stored into the array pointed to by **dst** (and *dst* is not a null pointer).

Each conversion takes place as if by a call to the **mbrtowc** function.

If *dst* is not a null pointer, the pointer object pointed to by src is assigned either a null pointer (if conversion stopped due to reaching a terminating null character) or the address just past the last character converted (if any). If conversion stopped due to reaching a terminating null character, and if *dst* is not a null pointer, the resulting state described is the initial conversion state.

If *ps* is a null pointer, the **mbsrtowcs** function uses its own internal **mbstate_t** object, which is initialised at program startup to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by ps is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls **mbsrtowcs**.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

If the input conversion encounters a sequence of bytes that do not form a valid character, an encoding error occurs. In this case, the **mbsrtowcs** function stores the value of the macro EILSEQ in errno and

returns (**size_t)-1**); the conversion state is undefined. Otherwise, it returns the number of characters successfully converted, not including the terminating null (if any).

Error Codes

The **mbsrtowcs** function may fail if:

Item	Description
EINVAL	ps points to an object that contains an invalid conversion state.
EILSEQ	Invalid character sequence is detected.

mbstomb Subroutine

Purpose

Extracts a multibyte character from a multibyte character string.

Note: The **mbstomb** subroutine is specific to the manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>

mbchar_t mbstomb ( MbString)
const char *MbString;
```

Description

The **mbstomb** subroutine extracts the multibyte character pointed to by the *MbString* parameter from the multibyte character string. The **LC_CTYPE** category affects the behavior of the **mbstomb** subroutine.

Parameters

Item	Description
MbString	Contains a multibyte character string.

Return Values

The **mbstomb** subroutine returns the code point of the multibyte character as a **mbchar_t** data type. If an unusable multibyte character is encountered, a value of 0 is returned.

mbstowcs Subroutine

Purpose

Converts a multibyte character string to a wide character string.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

size_t mbstowcs(*WcString*, *String*, *Number***)**
wchar_t ***_WcString_;**
const char ***_String_;**
size_t *Number***;**

Description

The **mbstowcs** subroutine converts the sequence of multibyte characters pointed to by the *String* parameter to wide characters and places the results in the buffer pointed to by the *WcString* parameter. The multibyte characters are converted until a null character is reached or until the number of wide characters specified by the *Number* parameter have been processed.

Parameters

Item	Description
WcString	Points to the area where the result of the conversion is stored.
String	Points to a multibyte character string.
Number	Specifies the maximum number of wide characters to be converted.

Return Values

The **mbstowcs** subroutine returns the number of wide characters converted, not including a null terminator, if any. If an invalid multibyte character is encountered, a value of -1 is returned. The *WcString* parameter does not include a null terminator if the value *Number* is returned.

If *WcString* is a null wide character pointer, the **mbstowcs** subroutine returns the number of elements required to store the wide character codes in an array.

Error Codes

The **mbstowcs** subroutine fails if the following occurs:

Item	Description
EILSEQ	Invalid byte sequence is detected.

mbswidth Subroutine

Purpose

Determines the number of multibyte character string display columns.

Note: The **mbswidth** subroutine is specific to this manufacturer. It is not defined in the POSIX, ANSI, or X/Open standards. Use of this subroutine may affect portability.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mbstr.h>
```

```
int mbswidth ( MbString,  Number)
const char *MbString;
size_t Number;
```

Description

The **mbswidth** subroutine determines the number of display columns required for a multibyte character string.

Parameters

Item	Description
MbString	Contains a multibyte character string.
Number	Specifies the number of bytes to read from the *s* parameter.

Return Values

The **mbswidth** subroutine returns the number of display columns that will be occupied by the *MbString* parameter if the number of bytes (specified by the *Number* parameter) read from the *MbString* parameter form valid multibyte characters. If the *MbString* parameter points to a null character, a value of 0 is returned. If the *MbString* parameter does not point to valid multibyte characters, -1 is returned. If the *MbString* parameter is a null pointer, the behavior of the **mbswidth** subroutine is unspecified.

mbtowc Subroutine

Purpose

Converts a multibyte character to a wide character.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

int mbtowc ( WideCharacter,  String,  Number)
wchar_t *WideCharacter;
const char *String;
size_t Number;
```

Description

The **mbtowc** subroutine converts a multibyte character to a wide character and returns the number of bytes of the multibyte character.

The **mbtowc** subroutine determines the number of bytes that comprise the multibyte character pointed to by the *String* parameter. It then converts the multibyte character to a corresponding wide character and, if the *WideCharacter* parameter is not a null pointer, places it in the location pointed to by the *WideCharacter* parameter. If the *WideCharacter* parameter is a null pointer, the **mbtowc** subroutine returns the number of converted bytes but does not change the *WideCharacter* parameter value. If the *WideCharacter* parameter returns a null value, the multibyte character is not converted.

Parameters

Item	Description
WideCharacter	Specifies the location where a wide character is to be placed.
String	Specifies a multibyte character.
Number	Specifies the maximum number of bytes of a multibyte character.

Return Values

The **mbtowc** subroutine returns a value of 0 if the *String* parameter is a null pointer. The subroutine returns a value of -1 if the bytes pointed to by the *String* parameter do not form a valid multibyte character before the number of bytes specified by the *Number* parameter (or fewer) have been processed. It then sets the **errno** global variable to indicate the error. Otherwise, the number of bytes comprising the multibyte character is returned.

Error Codes

The **mbtowc** subroutine fails if the following occurs:

Item	Description
EILSEQ	Invalid byte sequence is detected.

memccpy, memchr, memcmp, memcpy, memset, memset_s, or memmove Subroutine

Purpose

Performs memory operations and handles runtime constraint violations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <memory.h>
#include <string.h>
#define STDC_WANT_LIB_EXT1 1
```

```
void *memccpy (Target, Source, C, N)
void *Target;
const void *Source;
int C;
size_t N;
```

```
void *memchr ( S, C, N)
const void *S;
int C;
size_t N;
```

```
int memcmp (Target, Source, N)
const void *Target, *Source;
size_t N;
```

```
void *memcpy (Target, Source, N)
void *Target;
```

```
const void *Source;
size_t N;
```

```
void *memset (S, C, N)
void *S;
int C;
size_t N;
```

```
void *memmove (Target, Source,N)
void *Source;
const void *Target;
size_t N;
```

```
errno_t memset_s (s,smax,c,n)
void * s;
rsize_t smax;
int c;
rsize_t n;
```

Description

The **memory** subroutines operate on memory areas. A memory area is an array of characters bounded by a count. The **memory** subroutines do not check for the overflow of any receiving memory area. All of the **memory** subroutines are declared in the **memory.h** file.

The **memccpy** subroutine copies characters from the memory area specified by the *Source* parameter into the memory area specified by the *Target* parameter. The **memccpy** subroutine stops after the first character specified by the *C* parameter (converted to the **unsigned char** data type) is copied, or after *N* characters are copied, whichever comes first. If copying takes place between objects that overlap, the behavior is undefined.

The **memcmp** subroutine compares the first *N* characters as the **unsigned char**data type in the memory area specified by the *Target* parameter to the first *N* characters as the **unsigned char** data type in the memory area specified by the *Source* parameter.

The **memcpy** subroutine copies *N* characters from the memory area specified by the *Source* parameter to the area specified by the *Target* parameter and then returns the value of the *Target* parameter.

The **memset** subroutine sets the first *N* characters in the memory area specified by the *S* parameter to the value of character *C* and then returns the value of the *S* parameter.

Like the **memcpy** subroutine, the **memmove** subroutine copies *N* characters from the memory area specified by the *Source* parameter to the area specified by the *Target* parameter. However, if the areas of the *Source* and *Target* parameters overlap, the move is performed non-destructively, proceeding from right to left.

The **memccpy** subroutine is not in the ANSI C library.

The **memset_s** subroutine copies the value of **c** (converted to an unsigned character) into each of the first **n** characters of the object pointed by **s**. Unlike **memset**, any call to the **memset_s** function is evaluated according to the rules of the abstract machine and considers that the memory indicated by **s** and **n** might be accessible in the future and contains the values indicated by **c**.

Runtime Constraints

1. For the **memset_s** subroutine, the parameter **s** must not be a null pointer. Either **smax** or **n** can be greater than **RSIZE_MAX**, but **n** cannot be greater than **smax**.
2. If there is a runtime constraint violation and **s** is not a null pointer and **smax** is not greater than **RSIZE_MAX**, the **memset_s** subroutine stores the value of **c** (converted to an unsigned character) into each of the first **smax** characters of the object pointed by **s**.

Parameters

Item	Description
Target	Points to the start of a memory area.
Source	Points to the start of a memory area.
C	Specifies a character to search.
N	Specifies the number of characters to search.
S	Points to the start of a memory area.
s	Specifies the destination buffer for the copy.
c	Specifies the value to be copied.
smax	Specifies the maximum number of characters that can be copied.
n	Specifies the number of characters to be copied.

Return Values

The **memccpy** subroutine returns a pointer to character *C* after it is copied into the area specified by the *Target* parameter, or a null pointer if the *C* character is not found in the first *N* characters of the area specified by the *Source* parameter.

The **memchr** subroutine returns a pointer to the first occurrence of the *C* character in the first *N* characters of the memory area specified by the *S* parameter, or a null pointer if the *C* character is not found.

The **memcmp** subroutine returns the following values:

Item	Description
Less than 0	If the value of the *Target* parameter is less than the values of the *Source* parameter.
Equal to 0	If the value of the *Target* parameter equals the value of the *Source* parameter.
Greater than 0	If the value of the *Target* parameter is greater than the value of the *Source* parameter.

The **memset_s** subroutine returns zero if there is no runtime constraint violation. Otherwise, a nonzero value is returned.

meta Subroutine

Purpose

Enables/disables meta-keys.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int meta(WINDOW *win,
bool bf);
```

Description

Initially, whether the terminal returns 7 or 8 significant bits on input depends on the control mode of the display driver. To force 8 bits to be returned, invoke the **meta** subroutine (win, TRUE). To force 7 bits to be returned, invoke the **meta** subroutine (win, FALSE). The *win* argument is always ignored.

If the terminfo capabilities **smm** (meta_on) and **rmm** (meta_off) are defined for the terminal, **smm** is sent to the terminal when **meta** (win, TRUE) is called and **rmm** is sent when **meta** (win, FALSE) is called.

Parameters

Item	Description
bf	
**win*	

Return Values

Upon successful completion, the **meta** subroutine returns OK. Otherwise, it returns ERR.

Examples

1. To request an 8-bit character return when using a **getch** routine, enter:

```
WINDOW *some_window;
meta(some_window, TRUE);
```

2. To strip the highest bit off the character returns in the user-defined window `my_window`, enter:

```
WINDOW *some_window;
meta(some_window, FALSE);
```

mincore Subroutine

Purpose

Determines residency of memory pages.

Library

Standard C Library (**libc.a**).

Syntax

```
int mincore ( addr,  len, * vec)
caddr_t addr;
size_t len;
char *vec;
```

Description

The **mincore** subroutine returns the primary-memory residency status for regions created from calls made to the **mmap** subroutine. The status is returned as a character for each memory page in the range specified by the *addr* and *len* parameters. The least significant bit of each character returned is set to 1 if the referenced page is in primary memory. Otherwise, the bit is set to 0. The settings of the other bits in each character are undefined.

Parameters

Item	Description
addr	Specifies the starting address of the memory pages whose residency is to be determined. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
len	Specifies the length, in bytes, of the memory region whose residency is to be determined. If the *len* value is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region is rounded up to the next multiple of the page size.
vec	Specifies the character array where the residency status is returned. The system assumes that the character array specified by the *vec* parameter is large enough to encompass a returned character for each page specified.

Return Values

When successful, the **mincore** subroutine returns 0. Otherwise, it returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **mincore** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

Item	Description
EFAULT	A part of the buffer pointed to by the *vec* parameter is out of range or otherwise inaccessible.
EINVAL	The *addr* parameter is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
ENOMEM	Addresses in the (*addr*, *addr* + *len*) range are invalid for the address space of the process, or specify one or more pages that are not mapped.

MIO_aio_read64 Subroutine

Purpose

Read asynchronously from a file through MIO library.

Library

Modular I/O Library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_aio_read64( FileDescriptor, aiocbp )
```

```
int FileDescriptor;
struct aiocb64 *aiocbp;
```

Description

This subroutine is an entry point of the MIO library for the **Legacy AIO aio_read64** subroutine. Use this subroutine to instrument your application with the MIO library. You can replace the **Legacy AIO aio_read64 kernel I/O** subroutine with this equivalent MIO subroutine. See **Modular I/O** in *Performance management* for MIO library implementation.

Use this subroutine to read asynchronously from an open file specified by the *FileDescriptor* parameter. The *FileDescriptor* parameter results from an **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call aio_read64.

Return Values

The return values are those of the corresponding standard POSIX system call aio_read64.

Error Codes

The error codes are those of the corresponding standard POSIX system call aio_read64.

Location

/usr/lib/libmio.a

MIO_aio_suspend64 Subroutine

Purpose

Suspend the calling process until one or more asynchronous I/O requests are completed.

Library

Modular I/O Library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_aio_suspend64( Count, aiocbplist )
int Count;
struct aiocb64 **aiocbplist;
```

Description

This subroutine is an entry point of the MIO library for the **Legacy AIO aio_suspend64** subroutine. Use this subroutine to instrument your application with the MIO library. You can replace the **Legacy AIO aio_suspend64 kernel I/O** subroutine with this equivalent MIO subroutine. See **Modular I/O** in *Performance management* for the MIO library implementation.

The **aio_suspend64** subroutine suspends the calling process until one or more of the *Count* parameter asynchronous I/O requests are completed or a signal interrupts the subroutine. Specifically, the **aio_suspend64** subroutine handles requests associated with the aio control block (aiocb) structures pointed to by the *aiocbplist* parameter.

Parameters

The parameters are those of the corresponding standard POSIX system call aio_suspend64.

Return Values

The return values are those of the corresponding standard POSIX system call aio_suspend64.

Error Codes

The error codes are those of the corresponding standard POSIX system call aio_suspend64.

Location

/usr/lib/libmio.a

MIO_aio_write64 Subroutine

Purpose

Write asynchronously to a file through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_aio_write64( FileDescriptor, aiocbp )
int FileDescriptor;struct aiocb64 *aiocbp;
struct aiocb64 *aiocbp;
```

Description

This subroutine is an entry point of the MIO library for the **Legacy AIO aio_write64** subroutine. Use this subroutine to instrument your application with the MIO library. You can replace the **Legacy AIO aio_write64 kernel I/O** subroutine with this equivalent MIO subroutine. See **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to write asynchronously to an open file specified by the *FileDescriptor* parameter. The *FileDescriptor* parameter results from an **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call aio_write64.

Return Values

The return values are those of the corresponding standard POSIX system call aio_write64.

Error Codes

The error codes are those of the corresponding standard POSIX system call aio_write64.

Location

/usr/lib/libmio.a

MIO_close Subroutine

Purpose

Close a file descriptor through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_close (FileDescriptor)

int FileDescriptor;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the **close kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to close a file with the *FileDescriptor* parameter through the Modular I/O (MIO) library. The *FileDescriptor* parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call close.

Return Values

The return values are those of the corresponding standard POSIX system call close.

Error Codes

The error codes are those of the corresponding standard POSIX system call close.

Standard Output

MIO library outputs are flushed on the **MIO_close** subroutine call in the **stats** file.

The following is the information found in the diagnostic output file. It contains debug information:

- If you set the stats option of the trace module (trace/stats), it runs diagnostics from the trace module.
- If you set the stats option of the pf module (pf/stats), it runs diagnostics from the pf module.
- If you set the stats option of the recov module (recov/stats), it runs diagnostics from the recovery trace.
- If you set the nostats option of the trace or the pf module, these diagnostics are suppressed.

The diagnostic file name is defined in the MIO_STATS environment variable if the stats option is set to the default value of *mioout*.

To separate the trace, pf or recov module diagnostics from other outputs, set the stats options to the following other file names:

- trace/stats=<*tracefile*>
- pf/stats=<*pffile*>
- recov/stats=<*recovfile*>

The *tracefile*, *pffile* and *recovfile* are templates for the file names of module diagnostics output. You can give file names for the output of the trace, pf or recov module diagnostics.

Standard output includes the following information:

Header, which contains the following information:

- Date
- Hostname
- Enabled or disabled AIO
- Program name
- MIO library version
- Environment variables

Debug, which contains the following information:

- The list of all the debug options
- The table of all of the modules' definitions if the DEF debug option is set
- Open request made to the `MIO_open64` subroutine if the OPEN debug option is set
- The modules invoked if the MODULES debug option is set

Trace module diagnostic, which contains the following information:

- Time if the TIMESTAMP debug option is set
- Trace on close or on intermediate interrupt
- Trace module position in module_list
- Processed file name
- Rate, which is the amount of data divided by the total time. The total time here means the cumulative amount of time spent beneath the trace module
- Demand rate, which is the amount of data divided by the length of time when the file is opened (including the time of opening and closing the file)
- The current (when tracing) file size and the maximum size of the file during this file processing
- File system information: file type and sector size
- Open mode and flags of the file
- For each subroutine, the following information is displayed:

  name of the subroutine
  count of calling of this subroutine
  time of processing for this subroutine
- For read or write subroutines, the following information is displayed:

  requested (requested size to read or write) total (real size read or write: returned by AIX(r) system call) min (minimum size to read or write) max (maximum size to read or write)
- For the seek subroutine, the following information is displayed:

  the average seek delta (total seek delta/seek count)
- For the **aread** or **awrite** subroutine:

  count, time and rate of transfer time including suspend, and read or write time
- For the **fcntl** subroutine, the number of pages is returned.

The following is an example of a `trace` diagnostic:

date

```
Trace onclose or intermediate:
previous module or calling program<->next module:file name:
```

```
(total transferred bytes/total time)=rate
  demand rate=rate/s=total transferred bytes/(close time-open time)
  current size=actual size of the file
max_size=max size of the file
mode=file open mode
FileSystemType=file system type given by fststat(stat_b.f_vfstype)
sector size=Minimum direct i/o transfer size
oflags=file open flags
open     open count     open time
fcntl    fcntl count    fcntl time
read     read count     read time  requested size    total size   minimum    maximum
aread    aread count    aread time requested size    total size   minimum    maximum
suspend count time rate
write    write count    write time  requested size   total size   minimum    maximum
seek     seek count     seek time   average seek delta
size
page     fcntl page_info count
```

The following is a sample of a `trace` diagnostic:

```
MIO statistics file : Tue May 10 14:14:08 2005
hostname=host1  : with Legacy aio available
Program=example
MIO library libmio.a 3.0.0.60
Apr 19 2005 15:08:17
MIO_INSTALL_PATH=
MIO_STATS       =example.stats
MIO_DEBUG       =OPEN
MIO_FILES       = *.dat [ trace/stats ]
MIO_DEFAULTS    = trace/kbytes

MIO_DEBUG OPEN =T

Opening file file.dat
    modules[11]=trace/stats
================================================================================
Trace close : program <-> aix : file.dat : (4800/0.04)=111538.02 kbytes/s
      demand rate=42280.91 kbytes/s=4800/(0.12-0.01))
      current size=0   max_size=1600
   mode =0640  FileSystemType=JFS sector size=4096
   oflags =0x302=RDWR  CREAT  TRUNC
   open             1     0.00
   write          100     0.02      1600     1600     16384     16384
   read           200     0.02      3200     3200     16384     16384
   seek           101     0.01 average seek delta=-48503
   fcntl            1     0.00
   trunc            1     0.01
   close            1     0.00
   size           100
================================================================================
```

The following is a template of the `pf` module diagnostic:

pf close for<*name of the file in the cache*>
pf close for global or private cache <*global cache number*>
<*nb_pg_compute*>**page of**<*page-size*> <*sector_size*> **bytes per sector**
<*nb_real_pg_not_pf*>/<*nb_pg_not_pf*> **pages not preread for write**
<*nb_unused_pf*>**unused prefetches out of**<*nb_start_pf*>
prefetch=<*nb_pg_to_pf*>
<*number*> **of write behind**
<*number*> **of page syncs forced by ill formed writes**
<*number*> **of pages retained over close**
<*unit*> **transferred / Number of requests**
program --> <*bytes written into the cache by parent*>**/**
<*number of write from parent*>--> **pf** -->
<*written out of the cache from the child*>**/**<*number of partial page written*>
program --> <*bytes read out of the cache by parent*>**/**
<*number of read from parent*><- **pf** <-
<*bytes read in from child of the cache*>**/**<*number of page read from child*>

The following is explanation of the terms in the `pf` module template:

- *nb_pg_compute*= number of page compute by cache_size/ page size
- *nb_real_pg_not_pf*= real number page not prefetch because of *pffw* option (suppress number of page prefetch because sector not *valid*)
- *nb_pg_not_pf*= page of unused prefetch
- *nb_unused_pf*= number of started prefetch
- *nb_pg_to_pf*= number of page to prefetch

The following is a sample of the `pf` module diagnostic:

```
pf close for /home/user1/pthread/258/SM20182_0.SCR300
50 pages of 2097152 bytes 131072 bytes per sector
133/133 pages not preread for write
23 unused prefetches out of 242 : prefetch=2
95 write behinds
mbytes transferred / Number of requests
program --> 257/257 --> pf --> 257/131 --> aix
program <-- 269/269 <-- pf <-- 265/133 <-- aix
```

The following is the **recov** module output:

If open or write routine failed, the `recov` module, if set, is called. The `recov` module adds the following comments in the output file:

- The value of the *open_command* option
- The value of the *command* option
- The `errno`
- The index of retry

The following is a sample of the `recov` module:

```
15:30:00
   recov : command=ls -l file=file.dat errno=28 try=0
   recov : failure : new_ret=-1
```

Location

/usr/lib/libmio.a

MIO_fcntl Subroutine

Purpose

Control open file descriptors through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_fcntl ( FileDescriptor, Command, Argument )

int  FileDescriptor, Command, Argument;
```

Description

This subroutine is an entry point of the MIO library, offering the same features as the **fcntl** subroutine. Use this subroutine to instrument your application with the MIO library. You can replace the **fcntl kernel I/O** subroutine with this equivalent MIO subroutine. See **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to perform controlling operations on the open file specified by the *FileDescriptor* parameter. The *FileDescriptor* parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call fcntl.

Return Values

The return values are those of the corresponding standard POSIX system call fcntl.

Error Codes

The error codes are those of the corresponding standard POSIX system call fcntl.

Location

/usr/lib/libmio.a

MIO_ffinfo Subroutine

Purpose

Return file information through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>
int MIO_ffinfo (FileDescriptor, Command, Buffer, Length)
int FileDescriptor;
int Command;
struct diocapbuf  *Buffer;
int Length;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the **ffinfo kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for MIO library implementation.

Use this subroutine to obtain specific file information for the open file referenced by the *FileDescriptor* parameter. The *FileDescriptor* parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call ffinfo.

Return Values

The return values are those of the corresponding standard POSIX system call ffinfo.

Error Codes

The error codes are those of the corresponding standard POSIX system call ffinfo.

Location

/usr/lib/libmio.a

MIO_fstat64 Subroutine

Purpose

Provide information about a file through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>


int MIO_fstat64 (Filedescriptor, Buffer)
int FileDescriptor;
struct stat64 *Buffer;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the **fstat64 kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to obtain information about the open file referenced by *FileDescriptor* parameter. The *FileDescriptor* parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call fstat64.

Return Values

The return values are those of the corresponding standard POSIX system call fstat64.

Error Codes

The error codes are those of the corresponding standard POSIX system call fstat64.

Location

/usr/lib/libmio.a

MIO_fsync Subroutine

Purpose

Save changes in a file to permanent storage through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_fsync (FileDescriptor)

int FileDescriptor;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the fsync kernel I/O subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to save to permanent storage all modified data in the specified range of the open file specified by the *FileDescriptor* parameter. The *FileDescriptor* parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call fsync.

Return Values

The return values are those of the corresponding standard POSIX system call fsync.

Error Codes

The error codes are those of the corresponding standard POSIX system call fsync.

Location

/usr/lib/libmio.a

MIO_ftruncate64 Subroutine

Purpose

Change the length of regular files through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_ftruncate64 (FileDescriptor, Length)

int FileDescriptor;

int64 Length;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the **ftruncate64 kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to change the length of the open file specified by the *FileDescriptor* parameter through Modular I/O (MIO) library. The *FileDescriptor* parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call ftruncate64.

Return Values

The return values are those of the corresponding standard POSIX system call ftruncate64.

Error Codes

The error codes are those of the corresponding standard POSIX system call ftruncate64.

Location

/usr/lib/libmio.a

MIO_lio_listio64 Subroutine

Purpose

Initiate a list of asynchronous I/O requests with a single call.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_lio_listio64 (Command, List, Nent, Eventp)
int Command;
struct liocb64 *List;
int Nent;
struct event *Eventp;
```

Description

This subroutine is an entry point of the MIO library for the **Legacy AIO lio_listio64** Subroutine. Use this subroutine to instrument your application with MIO library. You can replace the **Legacy AIO lio_listio64**

kernel I/O subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

The **lio_listio64** subroutine allows the calling process to initiate the *Nent* parameter asynchronous I/O requests. These requests are specified in the liocb structures pointed to by the elements of the *List* array. The call may block or return immediately depending on the *Command* parameter. If the *Command* parameter requests that I/O completion be asynchronously notified, a SIGIO signal is delivered when all of the I/O operations are completed.

Parameters

The parameters are those of the corresponding standard POSIX system call lio_listio64.

Return Values

The return values are those of the corresponding standard POSIX system call lio_listio64.

Error Codes

The error codes are those of the corresponding standard POSIX system call lio_listio64.

Location

/usr/lib/libmio.a

MIO_lseek64 Subroutine

Purpose

Move the read-write file pointer through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>


int64 MIO_lseek64 (FileDescriptor, Offset, Whence)
int FileDescriptor;
int64 Offset;
int Whence;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the **fseek64 kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to set the read-write file pointer for the open file specified by the *FileDescriptor* parameter through the Modular I/O (MIO) library. The FileDescriptor parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call lseek64.

Return Values

The return values are those of the corresponding standard POSIX system call lseek64.

Error Codes

The error codes are those of the corresponding standard POSIX system call lseek64.

Location

/usr/lib/libmio.a

MIO_open64 Subroutine

Purpose

Opens a file for reading or writing through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_open64 (Path, OFlag, Mode, Extra)
char *Path;
int OFlag;int Mode;
struct mio_extra *Extra;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the **open64 kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to open a file through the Modular I/O (MIO) library. This library creates the context for this open file, according to the configuration set in MIO environment variables, or in the *Extra* parameter.

To analyze your application I/O and tune the I/O, use the MIO subroutines in the place of the standard I/O subroutines.

The MIO subroutines are:

- MIO_close
- MIO_lseek64
- MIO_read
- MIO_write
- MIO_ftruncate64
- MIO_fstat64
- MIO_fcntl
- MIO_ffinfo
- MIO_fsync

The standard I/O subroutines are:

- close

- lseek64
- read
- write
- ftruncate64
- fstat64
- fcntl
- finfo
- fsync

Parameters

Item	Description

Extra	Specifies some extra arguments for the MIO library. The simplest implementation is for any application to pass a zero pointer as the fourth argument. The fourth argument is a pointer to the mio_extra structure, you can usually pass a zero pointer, but you can also pass an mio_extra pointer (use this technique only if you are very familiar with how to code this argument).
	The mio_extra structure is defined in the following way:
	`struct mio_extra {`
	`int cookie ;`
	`/* Default value: MIO_EXTRA_COOKIE/`
	`int taskid ;`
	`/* for later */`
	`int64 bufsiz ;`
	`/* if > 1 : force the prefetch for write pffw */`
	`char *modules ;`
	`/* explicit module name,`
	`if any modules returns from MIO_FILES environment variable match */`
	`char *logical_name ;`
	`/* logical file name to open`
	`if file name don't match with MIO_FILES regexp`
	`*/`
	`int flags ;`
	`/* if MIO_EXTRA_SKIP_MIO_FILES_FLAG :`
	`don't use MIO_FILES env variable, but use extra->modules */`
	`} ;`

Mode	Specifies the modes. For more information, see the *Mode* flag in the **open64** subroutine.
Oflag	Specifies the type of access, the special open processing, the type of update, and the initial state of the open file. For more information, see the **open64** subroutine.
Path	Specifies the file to be opened.

Note: For applications that would not use the environment variable interface to apply the MIO modules to a file, the mio_extra hook provides an easy way to do that.

Environment variables

MIO is controlled by the following environment variables, which define the MIO features and are processed by the MIO_open64 subroutine:

The *MIO_STATS* variable is used to indicate a file that will be used as a repository for diagnostic messages and for output requested from the MIO modules. It is interpreted as a file name with two special cases. If the file is either the*stderr* or *stdout* output, the output will be directed towards the appropriate file stream. If the first character of the *MIO_STATS* variable is a plus sign (+), the file name to be used is the string following the plus sign (+), and the file is opened for appending. Without the preceding plus sign (+), the file is overwritten.

The *MIO_FILES* variable is the key to determine which modules are to be invoked for a given file when the MIO_open64 subroutine is called. The format for the *MIO_FILES* variable is the following:

```
first_file_name_list [ module list ] second_file_name_list [ module list] ...
```

When the MIO_open64 subroutine is called, MIO checks for the existence of the *MIO_FILES* variable and parses it as follows:

The *MIO_FILES* variable is parsed left to right. All characters up to the next occurrence of the bracket ([) are taken as a file name list. A file name list is a colon-separated list of file name templates. A file name template is used to match the name of the file opened by MIO and can use the following wildcard characters:

*****
Matches zero or more characters of a directory or file name.

?
Matches one character of a directory or file name.

******
Matches all remaining characters of a full path name.

If the file name templates does not contain a forward slash (/) , then all of the path directory information in the file name passed to the MIO_open64 subroutine is ignored and matching is applied only to the file name of the file being opened.

If the name of the file being opened is matched by one of the file name templates in the file name list then the module list to be invoked is taken as the string between brackets ([ ]). If the name of the file match two or more file name templates, the first match is taken into account. If the name of the file being opened does not match any of the file name templates in any of the file name lists then the file is opened with a default invocation of the AIX module.

If a match has occurred, the modules to be invoked are taken from the associated module list in the *MIO_FILES* variable. The modules are invoked left to right, with the left-most being closest to the user program and the right-most being closest to the operating system. If the module list does not start with the MIO module, a default invocation of the MIO module is added as a prefix. If the AIX module is not specified, a default invocation of the AIX module is appended.

The following is an example of the *MIO_FILES* variable:

```
setenv MIO_FILES " *.dat [ trace/stats ]"
```

Assume the *MIO_FILES* variable is set as follows:

```
MIO_FILES= *.dat:*.scr [ trace ] *.f01:*.f02:*.f03 [ trace | pf | trace ]
```

If the **test.dat** file is opened by the MIO_open64 subroutine, the **test.dat** file name matches ***.dat** and the following modules are invoked:

```
mio | trace | aix
```

If the **test.f02** file is opened by the MIO_open64 subroutine, the **test.f02** file name matches the second file name templates in the second file name list and the following modules are invoked:

```
mio | trace | pf | trace | aix
```

Each module has its own hardcoded default options for a default invocation. You can override the default options by specifying them in the associated *MIO_FILES* module list. The following example turns on the **stats** option for the trace module and requests that the output be directed to the **my.stats** file:

```
MIO_FILES= *.dat : *.scr [ trace/stats=my.stats ]
```

The options for a module are delimited with a forward slash (/). Some options require an associated string value and others might require an integer value. For those requiring a string value, if the string includes a forward slash (/), enclose the string in braces ({ }).

For those options requiring an integer value, append the integer value with a k, m, g, or t to represent kilo, mega, giga, or tera. You might also input integer values in base 10, 8, or 16. If you add a `0x` prefix to the integer value, the integer is interpreted as base 16. If you add a `0` prefix to the integer value, the integer is interpreted as base 8. If you add neither a `0x` prefix nor a `0` prefix to the integer value, the integer is interpreted as base 10.

The *MIO_DEFAULTS* variable is intended as a way to keep the *MIO_FILES* variable more readable. If the user is specifying several modules for multiple file name list and module list pairs, then the *MIO_FILES* variable might become quite long. To repeatedly override the hardcoded defaults in the same manner, you can specify new defaults for a module by specifying such defaults in the *MIO_DEFAULTS* variable. The *MIO_DEFAULTS* variable is a comma separated list of modules with their new defaults.

The following is an example of the *MIO_DEFAULTS* variable:

```
setenv MIO_DEFAULTS " trace/kbytes "
```

Assume that *MIO_DEFAULTS* variable is set as follows:

```
MIO_DEFAULTS = trace/events=prob.events , aix/debug
```

Any default invocation of the trace module will have binary event tracing enabled and directed towards the **prob.events** file and any default invocation of the AIX module will have debug enabled.

The *MIO_DEBUG* variable is intended as an aid in debugging the use of MIO. MIO searches the *MIO_DEFAULTS* variable for keywords and provides debugging output for the option. The available keywords are the following:

ALL
Turns on all of the *MIO_DEBUG* variable keywords.

ENV
Outputs environment variable matching requests.

OPEN
Outputs open requests made to the `MIO_open64` subroutine.

MODULES
Outputs modules invoked for each call to the `MIO_open64` subroutine.

TIMESTAMP
Places a timestamp preceding each entry into a **stats** file.

DEF
Outputs the definition table of each module. When the file opens, the outputs of all of the MIO library's definitions are processed for all the MIO library modules.

Return Values

The return values are those of the corresponding standard POSIX system call **open64**.

Error Codes

The error codes are those of the corresponding standard POSIX system call **open64**.

Standard Output

There is no MIO library output for the **MIO_open64** subroutine.

Note: MIO library output statistics are written in the **MIO_close** subroutine. This output filename is configurable with the *MIO_STATS* environment variable.

In the **example.stats** MIO output file, the module trace is set and reported, and the open requests are output. All of the values are in kilobytes.

Examples

The following **example.c** file issues 100 writes of 16 KB, seeks to the beginning of the file, issues 100 reads of 16 KB, and then seeks backward through the file reading 16 KB records. At the end the file is truncated to 0 bytes in length.

The *filename* argument to the following example is the file to be created, written to and read forwards and backwards:

```
------------------------------------------------------------------------------
#define _LARGE_FILES
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

#include "libmio.h"

/* Define open64, lseek64 and ftruncate64,  not
 * open, lseek, and ftruncate that are used in the code.  This is
 * because libmio.h defines _LARGE_FILES which forces <fcntl.h> to
 * redefine open, lseek, and ftruncate as open64, lseek64, and
 * ftruncate64
 */

#define open64(a,b,c) MIO_open64(a,b,c,0)
#define close         MIO_close
#define lseek64       MIO_lseek64
#define write         MIO_write
#define read          MIO_read
#define ftruncate64   MIO_ftruncate64

#define RECSIZE 16384
#define NREC    100

main(int argc, char **argv)
{
int i, fd, status ;
char *name ;
char *buffer ;
int64 ret64 ;

      if( argc < 2 ){
      fprintf(stderr,"Usage : example file_name\n");
      exit(-1);
   }
   name = argv[1] ;

   buffer = (char *)malloc(RECSIZE);
   memset( buffer, 0, RECSIZE ) ;

   fd = open(name, O_RDWR|O_TRUNC|O_CREAT, 0640 ) ;
   if( fd < 0 ){
      fprintf(stderr,"Unable to open file %s errno=%d\n",name,errno);
      exit(-1);
   }

/* write the file */
   for(i=0;i<NREC;i++){
      status = write( fd, buffer, RECSIZE ) ;
   }

/* read the file forwards */
   ret64 = lseek(fd, 0, SEEK_SET ) ;
   for(i=0;i<NREC;i++){
      status = read( fd, buffer, RECSIZE ) ;
   }
/* read the file backwards */
   for(i=0;i<NREC;i++){
      ret64 = lseek(fd, (NREC-i-1)*RECSIZE, SEEK_SET ) ;
      status = read( fd, buffer, RECSIZE ) ;
   }

/* truncate the file back to 0 bytes*/
   status = ftruncate( fd, 0 ) ;
```

```
   free(buffer);

/* close the file */
   status = close(fd);
}

-------------------------------------------------------------------------------
```

Both a script that sets the environment variables, compiles and calls the application and the **example.c** example are delivered and installed with the `libmio` file, as follows:

```
cc -o example example.c  -lmio

./example file.dat
```

The following environment variables are set to configure MIO:

setenv MIO_STATS example.stats
setenv MIO_FILES " *.dat [trace/stats] "
setenv MIO_DEFAULTS " trace/kbytes "
setenv MIO_DEBUG OPEN

See the **/usr/samples/libmio/README** file and sample files for details.

Location

/usr/lib/libmio.a

MIO_open Subroutine

Purpose

Opens a file for reading or writing through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_open (Path, OFlag, Mode, Extra)
char *Path;
int OFlag;
int Mode;
struct mio_extra *Extra;
```

Description

The **MIO_open** subroutine is a redirection to the `MIO_open64` subroutine and is an entry point of the MIO library. To use the MIO library, the files have to be opened with the **O_LARGEFILE** flag. For more details on the **O_LARGEFILE** flag, see the `fcntl.h` File.

Use the **MIO_open** subroutine to instrument your application with the MIO library. You can replace the **open kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to open a file through the Modular I/O (MIO) library. This library creates the context for this open file, according to the configuration set in the MIO environment variables, or in the *Extra* parameter.

To analyze your application I/O and tune the I/O, use the MIO subroutines in the place of the standard I/O subroutines.

The MIO subroutines are:

- MIO_close
- MIO_lseek64
- MIO_read
- MIO_write
- MIO_ftruncate64
- MIO_fstat64
- MIO_fcntl
- MIO_ffinfo
- MIO_fsync

The standard I/O subroutines are:

- close
- lseek64
- read
- write
- ftruncate64
- fstat64
- fcntl
- finfo
- fsync

Parameters

Item	Description

Extra	Specifies additional arguments for the MIO library. The simplest implementation is to pass a zero pointer as the fourth argument. The fourth argument is a pointer to the mio_extra structure, you can usually pass a zero pointer, but you can also pass an mio_extra pointer (use this technique only if you are very familiar with how to code this argument).
	The mio_extra structure is defined as follows:
	`struct mio_extra {`
	`int cookie ;`
	`/* Default value: MIO_EXTRA_COOKIE/`
	`int taskid ;`
	`/* for later */`
	`int64 bufsiz ;`
	`/* if > 1 : force the prefetch for write pffw */`
	`char *modules ;`
	`/* explicit module name,`
	`if any modules returns from MIO_FILES environment variable match */`
	`char *logical_name ;`
	`/* logical file name to open`
	`if file name don't match with MIO_FILES regexp`
	`*/`
	`int flags ;`
	`/* if MIO_EXTRA_SKIP_MIO_FILES_FLAG :`
	`don't use MIO_FILES env variable, but use extra->modules */`
	`} ;`

Mode	Specifies the modes. For more information, see the *Mode* flag in the **open64** subroutine.

Item	Description
Oflag	Specifies the type of access, the special open processing, the type of update, and the initial state of the open file. For more information, see the **open64** subroutine.
Path	Specifies the file to be opened.

Note: For applications that would not use the environment variable interface to apply MIO modules to a file, the mio_extra hook provides an easy way to do that.

Environment variables

MIO is controlled through the following four environment variables. These environment variables, which define the MIO features, are processed by the MIO_open64 subroutine.

The *MIO_STATS* variable is used to indicate a file that will be used as a repository for diagnostic messages and for output requested from the MIO modules. It is interpreted as a file name with two special cases. If the file is either the*stderr* or *stdout* output, the output will be directed towards the appropriate file stream. If the first character of the *MIO_STATS* variable is a plus sign (+), the file name to be used is the string following the plus sign (+), and the file is opened for appending. Without the preceding plus sign (+), the file is overwritten.

The *MIO_FILES* variable is the key to determine which modules are to be invoked for a given file when the MIO_open64 subroutine is called. The format for the *MIO_FILES* variable is the following:

```
first_file_name_list [ module list ] second_file_name_list [ module list]
```

When the MIO_open64 subroutine is called, MIO checks for the existence of the *MIO_FILES* variable and parses it as follows:

The *MIO_FILES* variable is parsed left to right. All characters up to the next occurrence of the bracket ([) are taken as a file name list. A file name list is a colon-separated list of file name templates. A file name template is used to match the name of the file opened by MIO and can use the following wildcard characters:

*****
Matches zero or more characters of a directory or file name.

?
Matches one character of a directory or file name.

******
Matches all remaining characters of a full path name.

If the file name template does not contain a forward slash (/) , then all of the path directory information in the file name passed to the MIO_open64 subroutine is ignored and matching is applied only to the file name of the file being opened.

If the name of the file being opened is matched by one of the file name templates in the file name list then the module list to be invoked is taken as the string between brackets ([ ]). If the name of the file match two or more file name templates, the first match is taken into account. If the name of the file being opened does not match any of the file name templates in any of the file name lists then the file is opened with a default invocation of the AIX module.

If a match has occurred, the modules to be invoked are taken from the associated module list in the *MIO_FILES* variable. The modules are invoked left to right, with the left-most being closest to the user program and the right-most being closest to the operating system. If the module list does not start with the MIO module, a default invocation of the MIO module is added as a prefix. If theAIX module is not specified, a default invocation of theAIX module is appended.

The following is an example of the *MIO_FILES* variable:

```
setenv MIO_FILES " *.dat [ trace/stats ]"
```

Assume the *MIO_FILES* variable is set as follows:

```
MIO_FILES= *.dat:*.scr [ trace ] *.f01:*.f02:*.f03 [ trace | pf | trace ]
```

If the **test.dat** file is opened by the MIO_open64 subroutine, the **test.dat** file name matches ***.dat** and the following modules are invoked:

```
mio | trace | aix
```

If the **test.f02** file is opened by the MIO_open64 subroutine, the **test.f02** file name matches the second file name templates in the second file name list and the following modules are invoked:

```
mio | trace | pf | trace | aix
```

Each module has its own hardcoded default options for a default invocation. You can override the default options by specifying them in the associated *MIO_FILES* module list. The following example turns on the **stats** option for the trace module and requests that the output be directed to the **my.stats** file:

```
MIO_FILES= *.dat : *.scr [ trace/stats=my.stats ]
```

The options for a module are delimited with a forward slash (/). Some options require an associated string value and others might require an integer value. For those requiring a string value, if the string includes a forward slash (/), enclose the string in braces ({ }).

For those options requiring an integer value, append the integer value with a k, m, g, or t to represent kilo, mega, giga, or tera. You might also input integer values in base 10, 8, or 16. If you add a 0x prefix to the integer value, the integer is interpreted as base 16. If you add a 0 prefix to the integer value, the integer is interpreted as base 8. If you add neither a 0x prefix nor a 0 prefix to the integer value, the integer is interpreted as base 10.

The *MIO_DEFAULTS* variable is intended as a way to keep the *MIO_FILES* variable more readable. If the user is specifying several modules for multiple file name list and module list pairs, then the *MIO_FILES* variable might become quite long. To repeatedly override the hardcoded defaults in the same manner, you can specify new defaults for a module by specifying such defaults in the *MIO_DEFAULTS* variable. The *MIO_DEFAULTS* variable is a comma separated list of modules with their new defaults.

The following is an example of the *MIO_DEFAULTS* variable:

```
setenv MIO_DEFAULTS " trace/kbytes "
```

Assume that *MIO_DEFAULTS* variable is set as follows:

```
MIO_DEFAULTS = trace/events=prob.events , aix/debug
```

Any default invocation of the trace module will have binary event tracing enabled and directed towards the **prob.events** file and any default invocation of the AIX module will have debug enabled.

The *MIO_DEBUG* variable is intended as an aid in debugging the use of MIO. MIO searches the *MIO_DEFAULTS* variable for keywords and provides debugging output for the option. The available keywords are the following:

ALL
Turns on all of the *MIO_DEBUG* variable keywords.

ENV
Outputs environment variable matching requests.

OPEN
Outputs open requests made to the MIO_open64 subroutine.

MODULES
Outputs modules invoked for each call to the MIO_open64 subroutine.

TIMESTAMP
Places a timestamp preceding each entry into a **stats** file.

DEF
Outputs the definition table of each module. When the file opens, the outputs of all of the MIO library's definitions are processed for all the MIO library modules.

Return values

The return values are those of the corresponding standard POSIX system call **open64**.

Error codes

The error codes are those of the corresponding standard POSIX system call **open64**.

Standard output

There is no MIO library output for the **MIO_open64** subroutine.

MIO library output statistics are written in the **MIO_close** subroutine. This output filename is configurable with the *MIO_STATS* environment variable.

In the **example.stats**. MIO output file, the module trace is set and reported, and the open requests are output. All the values are in kilobytes.

Examples

The following **example.c** file issues 100 writes of 16 KB, seeks to the beginning of the file, issues 100 reads of 16 KB, and then seeks backward through the file reading 16 KB records. At the end the file is truncated to 0 bytes in length.

The *filename* argument to the following example is the file to be created, written to and read forwards and backwards:

```
------------------------------------------------------------------------------
#define _LARGE_FILES
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>

#include "libmio.h"

/* Define open64, lseek64 and ftruncate64,  not
 * open, lseek, and ftruncate that are used in the code.  This is
 * because libmio.h defines _LARGE_FILES which forces <fcntl.h> to
 * redefine open, lseek, and ftruncate as open64, lseek64, and
 * ftruncate64
 */

#define open64(a,b,c) MIO_open64(a,b,c,0)
#define close         MIO_close
#define lseek64       MIO_lseek64
#define write         MIO_write
#define read          MIO_read
#define ftruncate64   MIO_ftruncate64

#define RECSIZE 16384
#define NREC    100

main(int argc, char **argv)
{
int i, fd, status ;
char *name ;
char *buffer ;
int64 ret64 ;

      if( argc < 2 ){
      fprintf(stderr,"Usage : example file_name\n");
      exit(-1);
   }
   name = argv[1] ;

   buffer = (char *)malloc(RECSIZE);
   memset( buffer, 0, RECSIZE ) ;
```

```
   fd = open(name, O_RDWR|O_TRUNC|O_CREAT, 0640 ) ;
   if( fd < 0 ){
      fprintf(stderr,"Unable to open file %s errno=%d\n",name,errno);
      exit(-1);
   }

/* write the file */
   for(i=0;i<NREC;i++){
      status = write( fd, buffer, RECSIZE ) ;
   }

/* read the file forwards */
   ret64 = lseek(fd, 0, SEEK_SET ) ;
   for(i=0;i<NREC;i++){
      status = read( fd, buffer, RECSIZE ) ;
   }
/* read the file backwards */
   for(i=0;i<NREC;i++){
      ret64 = lseek(fd, (NREC-i-1)*RECSIZE, SEEK_SET ) ;
      status = read( fd, buffer, RECSIZE ) ;
   }

/* truncate the file back to 0 bytes*/
   status = ftruncate( fd, 0 ) ;

   free(buffer);

/* close the file */
   status = close(fd);
}

--------------------------------------------------------------------------------
```

Both a script that sets the environment variables, compiles and calls the application and the **example.c** example are delivered and installed with the `libmio`, as follows:

```
cc -o example example.c  -lmio

./example file.dat
```

The following environment variables are set to configure MIO:

setenv MIO_STATS example.stats
setenv MIO_FILES " *.dat [trace/stats] "
setenv MIO_DEFAULTS " trace/kbytes "
setenv MIO_DEBUG OPEN

See the **/usr/samples/libmio/README** and sample files for details.

Location

/usr/lib/libmio.a

MIO_read Subroutine

Purpose

Read from a file through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_read(FileDescriptor,
```

```
Buffer, NBytes)
int  FileDescriptor;
void * Buffer;
int NBytes;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the **read kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to read to the number of bytes of data specified by the *NBytes* parameter from the file associated with the *FileDescriptor* parameter into the buffer, through the Modular I/O (MIO) library. The *Buffer* parameter points to the buffer. The *FileDescriptor* parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call read.

Return Values

The return values are those of the corresponding standard POSIX system call read.

Error Codes

The error codes are those of the corresponding standard POSIX system call read.

Location

/usr/lib/libmio.a

MIO_write Subroutine

Purpose

Write to a file through the MIO library.

Library

Modular I/O library (**libmio.a**)

Syntax

```
#include <libmio.h>

int MIO_write(FileDescriptor,
Buffer, NBytes)
int  FileDescriptor;
void * Buffer;
int  NBytes;
```

Description

This subroutine is an entry point of the MIO library. Use this subroutine to instrument your application with the MIO library. You can replace the **write kernel I/O** subroutine with this equivalent MIO subroutine. See the **Modular I/O** in *Performance management* for the MIO library implementation.

Use this subroutine to write the number of bytes of data specified by the *NBytes* parameter from the buffer to the file associated with the *FileDescriptor* parameter through the Modular I/O (MIO) library. The *Buffer* parameter points to the buffer. The *FileDescriptor* parameter results from the **MIO_open64** subroutine.

Parameters

The parameters are those of the corresponding standard POSIX system call write.

Return Values

The return values are those of the corresponding standard POSIX system call write.

Error Codes

The error codes are those of the corresponding standard POSIX system call write.

Location

/usr/lib/libmio.a

mkdir or mkdirat Subroutine

Purpose

Creates a directory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/stat.h>

int mkdir (Path,  Mode)
const char *Path;
mode_t Mode;

int mkdirat (DirFileDescriptor, Path, Mode)
int DirFileDescriptor;
const char * Path;
mode_t Mode;
```

Description

The **mkdir** and **mkdirat** subroutines create a new directory.

The new directory has the following:

- The owner ID is set to the process-effective user ID.
- If the parent directory has the *SetFileGroupID* (**S_ISGID**) attribute set, the new directory inherits the group ID of the parent directory. Otherwise, the group ID of the new directory is set to the effective group ID of the calling process.
- Permission and attribute bits are set according to the value of the *Mode* parameter, with the following modifications:
  - All bits set in the process-file mode-creation mask are cleared.

  - The *SetFileUserID* and *Sticky* (**S_ISVTX**) attributes are cleared.
- If the *Path* variable names a symbolic link, the link is followed. The new directory is created where the variable pointed.

The **mkdirat** subroutine is equivalent to the **mkdir** subroutine if the *DirFileDescriptor* parameter is set to **AT_FDCWD** or the *Path* parameter is an absolute path name. If the *DirFileDescriptor* parameter is a valid file descriptor of an open directory and the *Path* parameter is a relative path name, the *Path* parameter is considered as the relative path to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory.

If the *DirFileDescriptor* parameter is opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted for that directory by using the current permissions of the directory. If the directory is opened with the **O_SEARCH** open flag, the subroutine does not perform the check for that directory.

Parameters

Item	Description
Path	Specifies the name of the new directory. If Network File System (NFS) is installed on your system, this path can cross into another node. In this case, the new directory is created at that node. To execute the **mkdir** or **mkdirat** subroutine successfully, a process must have write permission to the parent directory of the *Path* parameter.
Mode	Specifies the mask for the read, write, and execute flags for owner, group, and others. The *Mode* parameter specifies directory permissions and attributes. This parameter is constructed by logically ORing values described in the **<sys/mode.h>** file.
DirFileDescriptor	Specifies the file descriptor of an open directory.

Return Values

Upon successful completion, the **mkdir** and **mkdirat** subroutines return a value of 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **mkdir** and **mkdirat** subroutines are unsuccessful and the directory is not created if one or more of the following are true:

Item	Description
EACCES	Creating the requested directory requires writing in a directory with a mode that denies write permission.
EEXIST	The named file already exists.
EROFS	The named file resides on a read-only file system.
ENOSPC	The file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.
EMLINK	The link count of the parent directory exceeds the maximum **(LINK_MAX)** number. **(LINK_MAX)** is defined in **limits.h** file.
ENAMETOOLONG	The *Path* parameter or a path component is too long and cannot be truncated.

Item	Description
ENOENT	A component of the path prefix does not exist or the *Path* parameter points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.
EDQUOT	The directory in which the entry for the new directory is being placed cannot be extended, or an i-node or disk blocks could not be allocated for the new directory because the user's or group's quota of disk blocks or i-nodes on the file system containing the directory is exhausted.

The **mkdirat** subroutine is unsuccessful if one or more of the following settings are true:

Item	Description
EBADF	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a valid file descriptor.
ENOTDIR	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.

The **mkdir** and **mkdirat** subroutines can be unsuccessful for other reasons. See "Appendix A. Base Operating System Error Codes for Services That Require Path-Name Resolution" for a list of additional errors.

If NFS is installed on the system, the **mkdir** and **mkdirat** subroutines are also unsuccessful if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

mknod, mknodat, mkfifo or mkfifoat, Subroutine

Purpose

Creates an ordinary file, first-in-first-out (FIFO), or special file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/stat.h>
```

```
int mknod (const char * Path, mode_t  Mode, dev_tDevice)
char *Path;
int Mode;
dev_t Device;
```

```
int mknodat (int DirFileDescriptor, const char * Path, mode_tMode, dev_tDevice)
int DirFileDescriptor;
char *Path;
int Mode;
dev_t Device;
```

```
int mkfifo (const char *Path, mode_t Mode)
const char *Path;
int Mode;
```

```
int mkfifoat (int DirFileDescriptor, const char *Path, mode_tMode)
int DirFileDescriptor;
const char *Path;
int Mode;
```

Description

The **mknod** and **mknodat** subroutines create a new regular file, special file, or FIFO file. Using the **mknod** or **mknodat** subroutine to create file types (other than FIFO or special files) requires root user authority.

For the **mknod** or **mknodat** subroutine to complete successfully, a process must have both search and write permission in the parent directory of the *Path* parameter.

The **mkfifo** and **mkfifoat** subroutines are interfaces to the **mknod** subroutine, where the new file to be created is a FIFO or special file. No special system privileges are required.

The new file has the following characteristics:

- File type is specified by the *Mode* parameter.
- Owner ID is set to the effective user ID of the process.
- Group ID of the file is set to the group ID of the parent directory if the *SetGroupID* attribute (**S_ISGID**) of the parent directory is set. Otherwise, the group ID of the file is set to the effective group ID of the calling process.
- Permission and attribute bits are set according to the value of the *Mode* parameter. All bits set in the file-mode creation mask of the process are cleared.

Upon successful completion, the **mkfifo** subroutine marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the file. It also marks for update the `st_ctime` and `st_mtime` fields of the directory that contains the new entry.

If the new file is a character special file having the **S_IMPX** attribute (multiplexed character special file), when the file is used, additional path-name components can appear after the path name as if it were a directory. The additional part of the path name is available to the device driver of the file for interpretation. This feature provides a multiplexed interface to the device driver.

The **mknodat** subroutine is equivalent to the **mknod** subroutine, and the **mkfifoat** subroutine is equivalent to the **mkfifo** subroutine if the *DirFileDescriptor* parameter is **AT_FDCWD** or *Path* is an absolute path name. If *DirFileDescriptor* is a valid file descriptor of an open directory and *Path* is a relative path name, *Path* is considered to be relative to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory.

If *DirFileDescriptor* was opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted for that directory by using the current permissions of the directory. If the directory was opened with the **O_SEARCH** open flag, the subroutine does not perform the check for that directory.

Parameters

Item	Description
DirFileDescriptor	Specifies the file descriptor of an open directory.
Path	Names the new file. If Network File System (NFS) is installed on your system, this path can cross into another node. If *DirFileDescriptor* is specified and *Path* is a relative path name, then *Path* is considered relative to the directory specified by *DirFileDescriptor*.

Item	Description
Mode	Specifies the file type, attributes, and access permissions. This parameter is constructed by logically ORing values described in the **<sys/mode.h>** file.
Device	Specifies the ID of the device, which corresponds to the `st_rdev` member of the structure returned by the **statx** subroutine. This parameter is configuration-dependent and used only if the *Mode* parameter specifies a block or character special file. If the file you specify is a remote file, the value of the *Device* parameter must be meaningful on the node where the file resides.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **mknod** and **mknodat** subroutines fail and the new file is not created if one or more of the following are true:

Item	Description
EEXIST	The named file exists.
EDQUOT	The directory in which the entry for the new file is being placed cannot be extended, or an i-node could not be allocated for the file because the user's or group's quota of disk blocks or i-nodes on the file system is exhausted.
EISDIR	The *Mode* parameter specifies a directory. Use the **mkdir** subroutine instead.
ENOSPC	The directory that would contain the new file cannot be extended, or the file system is out of file-allocation resources.
EPERM	The *Mode* parameter specifies a file type other than **S_IFIFO**, and the calling process does not have root user authority.
EROFS	The directory in which the file is to be created is located on a read-only file system.

The **mknodat** and **mkfifoat** subroutines fail and the new file is not created if one or more of the following are true:

Item	Description
EBADF	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a valid file descriptor.
ENOTDIR	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.

The **mknod**, **mknodat**, **mkfifo**, and **mkfifoat** subroutines can be unsuccessful for other reasons. See "Appendix. A Base Operating System Error Codes for Services That Require Path-Name Resolution" for a list of additional errors.

If NFS is installed on the system, the subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

mktemp or mkstemp Subroutine

Purpose

Constructs a unique file name.

Libraries

Standard C Library (**libc.a**)

Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <stdlib.h>

char *mktemp ( Template)
char *Template;

int mkstemp ( Template)
char *Template;
```

Description

The **mktemp** subroutine replaces the contents of the string pointed to by the *Template* parameter with a unique file name.

Note: The **mktemp** subroutine creates a filename and checks to see if the file exist. It that file does not exist, the name is returned. If the user calls **mktemp** twice without creating a file using the name returned by the first call to **mktemp**, then the second **mktemp** call may return the same name as the first **mktemp** call since the name does not exist.

To avoid this, either create the file after calling **mktemp** or use the **mkstemp** subroutine. The **mkstemp** subroutine creates the file for you.

To get the BSD version of this subroutine, compile with Berkeley Compatibility Library (**libbsd.a**).

The **mkstemp** subroutine performs the same substitution to the template name and also opens the file for reading and writing.

In BSD systems, the **mkstemp** subroutine was intended to avoid a race condition between generating a temporary name and creating the file. Because the name generation in the operating system is more random, this race condition is less likely. BSD returns a file name of / (slash).

Former implementations created a unique name by replacing X's with the process ID and a unique letter.

Parameters

Item	Description
Template	Points to a string to be replaced with a unique file name. The string in the *Template* parameter is a file name with up to six trailing X's. Since the system randomly generates a six-character string to replace the X's, it is recommended that six trailing X's be used.

Return Values

Upon successful completion, the **mktemp** subroutine returns the address of the string pointed to by the *Template* parameter.

If the string pointed to by the *Template* parameter contains no X's, and if it is an existing file name, the *Template* parameter is set to a null character, and a null pointer is returned; if the string does not match any existing file name, the exact string is returned.

Upon successful completion, the **mkstemp** subroutine returns an open file descriptor. If the **mkstemp** subroutine fails, it returns a value of -1.

mlock and munlock Subroutine

Purpose

Locks or unlocks a range of process address space.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>

int mlock (addr, len)
const void *addr;
size_t len;

int munlock (addr, len)
const void *addr;
size_t len;
```

Description

The **mlock** subroutine causes those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes to be memory-resident until unlocked or until the process exits or executes another process image. If the starting address *addr* is not a multiple of PAGESIZE, it is rounded down to the lowest page boundary. The *len* is rounded up to a multiple of PAGESIZE.

The **munlock** subroutine unlocks those whole pages containing any part of the address space of the process starting at address *addr* and continuing for *len* bytes, regardless of how many times **mlock** has been called by the process for any of the pages in the specified range.

If any of the pages in the range specified in a call to the **munlock** subroutine are also mapped into the address spaces of other processes, any locks established on those pages by another process are unaffected by the call of this process to the **munlock** subroutine. If any of the pages in the range specified by a call to the **munlock** subroutine are also mapped into other portions of the address space of the calling process outside the range specified, any locks established on those pages through other mappings are also unaffected by this call.

Upon successful return from **mlock**, pages in the specified range are locked and memory-resident. Upon successful return from **munlock**, pages in the specified range are unlocked with respect to the address space of the process.

The calling process must have the root user authority to use this subroutine.

Parameters

Item	Description
addr	Specifies the address space of the process to be locked or unlocked.
len	Specifies the length in bytes of the address space.

Return Values

Upon successful completion, the **mlock** and **munlock** subroutines return zero. Otherwise, no change is made to any locks in the address space of the process, the surbroutines return -1 and set **errno** to indicate the error.

Error Codes

The **mlock** and **munlock** subroutines fail if:

Item	Description
ENOMEM	Some or all of the address range specified by the *addr* and *len* parameters does not correspond to valid mapped pages in the address space of the process.
EINVAL	The process has already some plocked memory or the *len* parameter is negative.
EPERM	The calling process does not have the appropriate privilege to perform the requested operation.

The **mlock** subroutine might fail if:

Item	Description
ENOMEM	Locking the pages mapped by the specified range would exceed the limit on the amount of memory the process may lock.

mlockall and munlockall Subroutine

Purpose

Locks or unlocks the address space of a process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>

int mlockall (flags)
int flags;

int munlockall (void);
```

Description

The **mlockall** subroutine causes all of the pages mapped by the address space of a process to be memory-resident until unlocked or until the process exits or executes another process image. The *flags* parameter determines whether the pages to be locked are those currently mapped by the address space of the process, those that are mapped in the future, or both. The *flags* parameter is constructed from the bitwise-inclusive OR of one or more of the following symbolic constants, defined in the **sys/mman.h** header file:

MCL_CURRENT
Lock all of the pages currently mapped into the address space of the process.

MCL_FUTURE
Lock all of the pages that become mapped into the address space of the process in the future, when those mappings are established.

When **MCL_FUTURE** is specified, the future mapping functions might fail if the system is not able to lock this amount of memory because of lack of resources, for example.

The **munlockall** subroutine unlocks all currently mapped pages of the address space of the process. Any pages that become mapped into the address space of the process after a call to the **munlockall** subroutine are not locked, unless there is an intervening call to the **mlockall** subroutine specifying **MCL_FUTURE** or a subsequent call to the **mlockall** subroutine specifying **MCL_CURRENT**. If pages mapped into the address space of the process are also mapped into the address spaces of other processes and are locked by those processes, the locks established by the other processes are unaffected by a call to the **munlockall** subroutine.

Regarding libraries that are pinned, a distinction has been made internally between a user referencing memory to perform a task related to the application and the system referencing memory on behalf of the application. The former is pinned, and the latter is not. The user-addressable loader data that remains unlocked includes:

- loader entries
- user loader entries
- page-descriptor segment
- usla heap segment
- usla text segment
- all the global segments related to the 64-bit shared library loadlist (shlib heap segment, shlib le segment, shlib text and data heap segments).

This limit affects implementation only, and it does not cause the API to fail.

Upon successful return from a **mlockall** subroutine that specifies **MCL_CURRENT**, all currently mapped pages of the process' address space are memory-resident and locked. Upon return from the **munlockall** subroutine, all currently mapped pages of the process' address space are unlocked with respect to the process' address space.

The calling process must have the root user authority to use this subroutine.

Parameters

Item	Description
flags	Determines whether the pages to be locked are those currently mapped by the address space of the process, those that are mapped in the future, or both.

Return Values

Upon successful completion, the **mlockall** subroutine returns 0. Otherwise, no additional memory is locked, and the subroutine returns -1 and sets **errno** to indicate the error.

Upon successful completion, the **munlockall** subroutine returns 0. Otherwise, no additional memory is unlocked, and the subroutine returns -1 and sets **errno** to indicate the error.

Error Codes

The **mlockall** subroutine fails if:

Item	Description
EINVAL	The *flags* parameter is 0, or includes unimplemented flags or the process has already some plocked memory.

Item	Description
ENOMEM	Locking all of the pages currently mapped into the address space of the process would exceed the limit on the amount of memory that the process may lock.
EPERM	The calling process does not have the appropriate authority to perform the requested operation.

The **munlockall** subroutine fails if:

Item	Description
EINVAL	The process has already some plocked memory
EPERM	The calling process does not have the appropriate privilege to perform the requested operation

mmap or mmap64 Subroutine

Purpose

Maps a file-system object into virtual memory.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>
```

```
void *mmap (addr, len, prot, flags, fildes, off)
void * addr;
size_t  len;
int  prot,  flags,  fildes;
off_t  off;
```

```
void *mmap64 (addr, len, prot, flags, fildes, off)
void * addr;
size_t  len;
int  prot,  flags,  fildes;
off64_t  off;
```

Description

Attention: A file-system object should not be simultaneously mapped using both the **mmap** and **shmat** subroutines. Unexpected results may occur when references are made beyond the end of the object.

The **mmap** subroutine creates a new mapped file or anonymous memory region by establishing a mapping between a process-address space and a file-system object. Care needs to be taken when using the **mmap** subroutine if the program attempts to map itself. If the page containing executing instructions is currently referenced as data through an mmap mapping, the program will hang. Use the -H4096 binder option, and that will put the executable text on page boundaries. Then reset the file that contains the executable material, and view via an **mmap** mapping.

A region created by the mmap subroutine cannot be used as the buffer for read or write operations that involve a device. Similarly, an mmap region cannot be used as the buffer for operations that require either a **pin** or **xmattach** operation on the buffer.

Modifications to a file-system object are seen consistently, whether accessed from a mapped file region or from the **read** or **write** subroutine.

Child processes inherit all mapped regions from the parent process when the **fork** subroutine is called. The child process also inherits the same sharing and protection attributes for these mapped regions. A successful call to any **exec** subroutine will unmap all mapped regions created with the **mmap** subroutine.

The **mmap64** subroutine is identical to the **mmap** subroutine except that the starting offset for the file mapping is specified as a 64-bit value. This permits file mappings which start beyond **OFF_MAX**.

In the large file enabled programming environment, **mmap** is redefined to be **mmap64**.

If the application has requested SPEC1170 compliant behavior then the **st_atime** field of the mapped file is marked for update upon successful completion of the **mmap** call.

If the application has requested SPEC1170 compliant behavior then the **st_ctime** and **st_mtime** fields of a file that is mapped with **MAP_SHARED** and **PROT_WRITE** are marked for update at the next call to **msync** subroutine or **munmap** subroutine if the file has been modified.

Parameters

Item	Description
addr	Specifies the starting address of the memory region to be mapped. When the **MAP_FIXED** flag is specified, this address must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter. A region is never placed at address zero, or at an address where it would overlap an existing region.
len	Specifies the length, in bytes, of the memory region to be mapped. The system performs mapping operations over whole pages only. If the *len* parameter is not a multiple of the page size, the system will include in any mapping operation the address range between the end of the region and the end of the page containing the end of the region.

Item	Description
prot	Specifies the access permissions for the mapped region. The **sys/mman.h** file defines the following access options: **PROT_READ** Region can be read. **PROT_WRITE** Region can be written. **PROT_EXEC** Region can be executed. **PROT_NONE** Region cannot be accessed. The *prot* parameter can be the **PROT_NONE** flag, or any combination of the **PROT_READ** flag, **PROT_WRITE** flag, and **PROT_EXEC** flag logically ORed together. If the **PROT_NONE** flag is not specified, access permissions may be granted to the region in addition to those explicitly requested. However, write access will not be granted unless the **PROT_WRITE** flag is specified. **Note:** The operating system generates a **SIGSEGV** signal if a program attempts an access that exceeds the access permission given to a memory region. For example, if the **PROT_WRITE** flag is not specified and a program attempts a write access, a **SIGSEGV** signal results. If the region is a mapped file that was mapped with the **MAP_SHARED** flag, the **mmap** subroutine grants read or execute access permission only if the file descriptor used to map the file was opened for reading. It grants write access permission only if the file descriptor was opened for writing. If the region is a mapped file that was mapped with the **MAP_PRIVATE** flag, the **mmap** subroutine grants read, write, or execute access permission only if the file descriptor used to map the file was opened for reading. If the region is an anonymous memory region, the **mmap** subroutine grants all requested access permissions.

Item	Description
flags	Specifies attributes of the mapped region. Values for the *flags* parameter are constructed by a bitwise-inclusive ORing of values from the following list of symbolic names defined in the **sys/mman.h** file: **MAP_FILE** Specifies the creation of a new mapped file region by mapping the file associated with the *fildes* file descriptor. The mapped region can extend beyond the end of the file, both at the time when the **mmap** subroutine is called and while the mapping persists. This situation could occur if a file with no contents was created just before the call to the **mmap** subroutine, or if a file was later truncated. However, references to whole pages following the end of the file result in the delivery of a **SIGBUS** signal. Only one of the **MAP_FILE** and **MAP_ANONYMOUS** flags must be specified with the **mmap** subroutine. **MAP_ANONYMOUS** Specifies the creation of a new, anonymous memory region that is initialized to all zeros. This memory region can be shared only with the descendants of the current process. When using this flag, the *fildes* parameter must be -1. Only one of the **MAP_FILE** and **MAP_ANONYMOUS** flags must be specified with the **mmap** subroutine. **MAP_ VARIABLE** Specifies that the system select an address for the new memory region if the new memory region cannot be mapped at the address specified by the *addr* parameter, or if the *addr* parameter is null. Only one of the **MAP_VARIABLE** and **MAP_FIXED** flags must be specified with the **mmap** subroutine. **MAP_FIXED** Specifies that the mapped region be placed exactly at the address specified by the *addr* parameter. If the application has requested SPEC1170 complaint behavior and the **mmap** request is successful, the mapping replaces any previous mappings for the process' pages in the specified range. If the application has not requested SPEC1170 compliant behavior and a previous mapping exists in the range then the request fails. Only one of the **MAP_VARIABLE** and **MAP_FIXED** flags must be specified with the **mmap** subroutine. **MAP_SHARED** When the **MAP_SHARED** flag is set, modifications to the mapped memory region will be visible to other processes that have mapped the same region using this flag. If the region is a mapped file region, modifications to the region will be written to the file. You can specify only one of the **MAP_SHARED** or **MAP_PRIVATE** flags with the **mmap** subroutine. **MAP_PRIVATE** is the default setting when neither flag is specified unless you request SPEC1170 compliant behavior. In this case, you must choose either **MAP_SHARED** or **MAP_PRIVATE**. **MAP_PRIVATE** When the **MAP_PRIVATE** flag is specified, modifications to the mapped region by the calling process are not visible to other processes that have mapped the same region. If the region is a mapped file region, modifications to the region are not written to the file. If this flag is specified, the initial write reference to an object page creates a private copy of that page and redirects the mapping to the copy. Until then, modifications to the page by processes that have mapped the same region with the **MAP_SHARED** flag are visible. You can specify only one of the **MAP_SHARED** or **MAP_PRIVATE** flags with the **mmap** subroutine. **MAP_PRIVATE** is the default setting when neither flag is specified unless you request SPEC1170 compliant behavior. In this case, you must choose either **MAP_SHARED** or **MAP_PRIVATE**.

Item	Description
fildes	Specifies the file descriptor of the file-system object or of the shared memory object to be mapped. If the **MAP_ANONYMOUS** flag is set, the *fildes* parameter must be -1. After the successful completion of the **mmap** subroutine, the file or the shared memory object specified by the *fildes* parameter can be closed without affecting the mapped region or the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, which prevents the file data from being deallocated. **Note:** The **mmap** subroutine supports the mapping of shared memory object and regular files only. An **mmap** call that specifies a file descriptor for a special file fails, returning the **ENODEV** error code. An example of a file descriptor for a special file is one that might be used for mapping either I/O or device memory.
off	Specifies the file byte offset at which the mapping starts. This offset must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.

Return Values

If successful, the **mmap** subroutine returns the address at which the mapping was placed. Otherwise, it returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

Under the following conditions, the **mmap** subroutine fails and sets the **errno** global variable to:

Item	Description
EACCES	The file referred to by the *fildes* parameter is not open for read access, or the file is not open for write access and the **PROT_WRITE** flag was specified for a **MAP_SHARED** mapping operation. Or, the file to be mapped has enforced locking enabled and the file is currently locked.
EAGAIN	The *fildes* parameter refers to a device that has already been mapped.
EBADF	The *fildes* parameter is not a valid file descriptor, or the **MAP_ANONYMOUS** flag was set and the *fildes* parameter is not -1.
EFBIG	The mapping requested extends beyond the maximum file size associated with *fildes*.
EINVAL	The *flags* or *prot* parameter is invalid, or the *addr* parameter or *off* parameter is not a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
EINVAL	The application has requested SPEC1170 compliant behavior and the value of flags is invalid (neither **MAP_PRIVATE** nor **MAP_SHARED** is set).
EMFILE	The application has requested SPEC1170 compliant behavior and the number of mapped regions would excedd and implementation-dependent limit (per process or per system).
ENODEV	The *fildes* parameter refers to an object that cannot be mapped, such as a terminal.
ENOMEM	There is not enough address space to map *len* bytes, or the application has not requested Single UNIX Specification, Version 2 compliant behavior and the **MAP_FIXED** flag was set and part of the address-space range (*addr*, *addr+len*) is already allocated.
ENXIO	The addresses specified by the range (*off*, *off+len*) are invalid for the *fildes* parameter.

Item	Description
EOVERFLOW	The mapping requested extends beyond the offset maximum for the file description associated with *fildes*.

mmcr_read Subroutine

Purpose

Reads the monitor mode control registers **MMCR0**, **MMCR2**, and **MMCRA** in problem state.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int mmcr_read (void * buffer)
```

Description

The **mmcr_read** subroutine reads the registers **MMCR0**, **MMCR2**, and **MMCRA** in the same order into the address of the buffer that is passed as a parameter to the function.

The three 64-bit **MMCR** registers **MMCR0**, **MMCR2**, and **MMCRA** are read into the buffer.

Return Values

If unsuccessful, a value of other than zero is returned and positive error code is set. If successful, a value of zero is returned and no errors are detected.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

mmcr_write Subroutine

Purpose

Writes the specified monitor mode control register in problem state.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int mmcr_write (int reg_num, void *buffer)
```

Description

The **mmcr_write** subroutine writes a specified monitor mode control register (MMCR) in problem state.

The function takes two parameters namely the Special Purpose Register (SPR) number of the MMCR into which the value is written, and the address from where the value is written to the MMCR. The **mmcr_write**

subroutine writes the value of the address specified in the second argument into the register specified in the first argument.

Return Values

If unsuccessful, a value other than zero is returned and positive error code is set. If successful, a value of zero is returned and no errors are detected.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

mntctl Subroutine

Purpose

Returns the mount status of file systems, or alters the status of mounted file systems.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/mntctl.h>
#include <sys/vmount.h>

int mntctl ( Command,  Size,  Buffer)
int Command;
int Size;
char *Buffer;
```

Description

The **mntctl** subroutine is used to query the status of virtual file systems (also known as *mounted* file systems). It can also be used to alter the state of mounted file systems.

Each virtual file system (VFS) is described by a **vmount** structure. This structure is supplied when the VFS is created by the **vmount** subroutine. The **vmount** structure is defined in the **sys/vmount.h** file.

Parameters

Item	Description
Command	Specifies the operation to be performed. Valid commands are defined in the **sys/vmount.h** file. At present, the only command is: **MCTL_QUERY** Query mount information. **MCTL_REMNT** Re-mount a mounted file system with the options specified in the **vmount** structure passed in. The **MCTL_REMNT** command is only passed to file systems that support the capability to re-mount. For more information, see the **gfsadd** Kernel Service.

Item	Description
Buffer	For the **MCTL_QUERY** command, the *Buffer* parameter points to a data area that will contain an array of the **vmount** structures. Because the **vmount** structure is variable-length, it is necessary to reference the **vmt_length** field of each structure to determine where in the *Buffer* area the next structure begins. For the **MCTL_REMNT** command, the *Buffer* parameter points to a data area that contains the **vmount** structure that is passed in.
Size	Specifies the length, in bytes, of the buffer pointed to by the *Buffer* parameter.

Return Values

For the **MCTL_QUERY** command, if the **mntctl** subroutine is successful, the number of **vmount** structures that are copied into the *Buffer* parameter is returned. If the *Size* parameter indicates that the supplied buffer is too small to hold the **vmount** structures for all of the current VFSs, the **mntctl** subroutine sets the first word of the *Buffer* parameter to the required size (in bytes) and returns the value of 0. If the **mntctl** subroutine otherwise fails, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

For the **MCTL_REMNT** command, if the **mntctl** subroutine fails, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **mntctl** subroutine fails and the requested operation is not performed if one or both of the following are true:

Item	Description
EINVAL	The *Command* parameter is not recognized, or the *Size* parameter is not a positive value.
EFAULT	The *Buffer* parameter points to a location outside of the allocated address space of the process.

modf, modff, modfl, modfd32, modfd64, and modfd128 Subroutines

Purpose

Decomposes a floating-point number.

Syntax

```
#include <math.h>

float modff (x, iptr)
float x;
float *iptr;

double modf (x, iptr)
double x, *iptr;

long double modfl (x, iptr)
long double x, *iptr;

_Decimal32 modfd32 (x, iptr)
_Decimal32 x, *iptr;

_Decimal64 modfd64 (x, iptr)
_Decimal64 x, *iptr;
```

```
_Decimal128 modf128 (x, iptr)
_Decimal128 x, *iptr;
```

Description

The **modff**, **modf**, **modfl**, **modfd32**, **modfd64**, and **modfd128** subroutines divide the *x* parameter into integral and fractional parts, each of which has the same sign as the arguments. These subroutines store the integral part as a floating-point value in the object pointed to by the *iptr* parameter.

Parameters

Item	Description
x	Specifies the value to be computed.
iptr	Points to the object where the integral part is stored.

Return Values

Upon successful completion, the **modff**, **modf**, **modfl**, **modfd32**, **modfd64**, and **modfd128** subroutines return the signed fractional part of *x*.

If *x* is NaN, a NaN is returned, and **iptr* is set to a NaN.

If *x* is ±Inf, ±0 is returned, and **iptr* is set to ±Inf.

moncontrol Subroutine

Purpose

Starts and stops execution profiling after initialization by the **monitor** subroutine.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mon.h>
```

```
int moncontrol ( Mode)
int Mode;
```

Description

The **moncontrol** subroutine starts and stops profiling after profiling has been initialized by the **monitor** subroutine. It may be used with either **-p** or **-pg** profiling. When **moncontrol** stops profiling, no output data file is produced. When profiling has been started by the **monitor** subroutine and the **exit** subroutine is called, or when the **monitor** subroutine is called with a value of 0, then profiling is stopped, and an output file is produced, regardless of the state of profiling as set by the **moncontrol** subroutine.

The **moncontrol** subroutine examines global and parameter data in the following order:

1. When the **_mondata.prof_type** global variable is neither -1 (**-p** profiling defined) nor +1 (**-pg** profiling defined), no action is performed, 0 is returned, and the function is considered complete.

   The global variable is set to -1 in the **mcrt0.o** file and to +1 in the **gcrt0.o** file and defaults to 0 when the **crt0.o** file is used.
2. When the *Mode* parameter is 0, profiling is stopped. For any other value, profiling is started.

The following global variables are used in a call to the **profil** subroutine:

Item	Description
_mondata.*ProfBuf*	Buffer address
_mondata.*ProfBufSiz*	Buffer size/multirange flag
_mondata.*ProfLoPC*	PC offset for hist buffer - I/O limit
_mondata.*ProfScale*	PC scale/compute scale flag.

These variables are initialized by the **monitor** subroutine each time it is called to start profiling.

Parameters

Item	Description
Mode	Specifies whether to start (resume) or stop profiling.

Return Values

The **moncontrol** subroutine returns the previous state of profiling. When the previous state was STOPPED, a 0 is returned. When the previous state was STARTED, a 1 is returned.

Error Codes

When the **moncontrol** subroutine detects an error from the call to the **profil** subroutine, a -1 is returned.

monitor Subroutine

Purpose

Starts and stops execution profiling using data areas defined in the function parameters.

Library

Standard C Library (**libc.a**)

Syntax

#include <mon.h>int monitor (*LowProgramCounter***,***HighProgramCounter***,***Buffer***,***BufferSize***,***NFunction***)**OR **int monitor (***NotZeroA***,***DoNotCareA***, *Buffer*,**-1**, *NFunction***)**OR **int monitor((caddr_t)**0**)caddr_t** *LowProgramCounter***, *HighProgramCounter*;HISTCOUNTER *****Buffer*;**int** *BufferSize***, *NFunction*;caddr_t** *NotZeroA***, *DoNotCareA*;**

Description

The **monitor** subroutine initializes the buffer area and starts profiling, or else stops profiling and writes out the accumulated profiling data. Profiling, when started, causes periodic sampling and recording of the program location within the program address ranges specified. Profiling also accumulates function call count data compiled with the **-p** or **-pg** option.

Executable programs created with the **cc -p** or **cc -pg** command automatically include calls to the **monitor** subroutine (through the **monstartup** and **exit** subroutines) to profile the complete user program, including system libraries. In this case, you do not need to call the **monitor** subroutine.

The **monitor** subroutine is called by the **monstartup** subroutine to begin profiling and by the **exit** subroutine to end profiling. The **monitor** subroutine requires a global data variable to define which kind of profiling, **-p** or **-pg**, is in effect. The **monitor** subroutine initializes four global variables that are used as parameters to the **profil** subroutine by the **moncontrol** subroutine:

- The **monitor** subroutine calls the **moncontrol** subroutine to start the profiling data gathering.
- The **moncontrol** subroutine calls the **profil** subroutine to start the system timer-driven program address sampling.
- The **prof** command processes the data file produced by **-p** profiling.
- The **gprof** command processes the data file produced by **-pg** profiling.

The **monitor** subroutine examines the global data and parameter data in this order:

1. When the **_mondata.prof_type** global variable is neither -1 (**-p** profiling defined) nor +1 (**-pg** profiling defined), an error is returned, and the function is considered complete.

   The global variable is set to -1 in the **mcrt0.o** file and to +1 in the **gcrt0.o** file, and defaults to 0 when the **crt0.o** file is used.
2. When the first parameter to the **monitor** subroutine is 0, profiling is stopped and the data file is written out.

   If **-p** profiling was in effect, then the file is named **mon.out**. If **-pg** profiling was in effect, the file is named **gmon.out**. The function is complete.
3. When the first parameter to the **monitor** subroutine is not , the **monitor** parameters and the profiling global variable, **_mondata.prof_type,** are examined to determine how to start profiling.
4. When the *BufferSize* parameter is not -1, a single program address range is defined for profiling, and the first **monitor** definition in the syntax is used to define the single program range.
5. When the *BufferSize* parameter is -1, multiple program address ranges are defined for profiling, and the second **monitor** definition in the syntax is used to define the multiple ranges. In this case, the *ProfileBuffer* value is the address of an array of **prof** structures. The size of the **prof** array is denoted by a zero value for the*HighProgramCounter* (`p_high`) field of the last element of the array. Each element in the array, except the last, defines a single programming address range to be profiled. Programming ranges must be in ascending order of the program addresses with ascending order of the **prof** array index. Program ranges may not overlap.

   The buffer space defined by the `p_buff` and `p_bufsize` fields of all of the **prof** entries must define a single contiguous buffer area. Space for the function-count data is included in the first range buffer. Its size is defined by the *NFunction* parameter. The `p_scale` entry in the **prof** structure is ignored. The **prof** structure is defined in the**mon.h** file. It contains the following fields:

```
caddr_t p_low;        /* low sampling address */
caddr_t p_high;       /* high sampling address */
HISTCOUNTER *p_buff;  /* address of sampling buffer */
int p_bufsize;        /* buffer size- monitor/HISTCOUNTERs,\
                         profil/bytes */
uint p_scale;         /* scale factor */
```

Parameters

Item	Description
LowProgramCounter (**prof** name: `p_low`)	Defines the lowest execution-time program address in the range to be profiled. The value of the *LowProgramCounter* parameter cannot be 0 when using the**monitor** subroutine to begin profiling.

Item	Description
HighProgramCounter (**prof** name: p_high)	Defines the next address after the highest-execution time program address in the range to be profiled. The program address parameters may be defined by function names or address expressions. If defined by a function name, then a function name expression must be used to dereference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use.
Buffer (**prof** name: p_buff)	Defines the beginning address of an array of *BufferSize* HISTCOUNTERs to be used for data collection. This buffer includes the space for the program address-sampling counters and the function-count data areas. In the case of a multiple range specification, the space for the function-count data area is included at the beginning of the first range in the *BufferSize* specification.
BufferSize (**prof** name: p_bufsize)	Defines the size of the buffer in number of HISTCOUNTERs. Each counter is of type HISTCOUNTER (defined as short in the **mon.h** file). When the buffer includes space for the function-count data area (single range specification and first range of a multi-range specification) the *NFunction* parameter defines the space to be used for the function count data, and the remainder is used for program-address sampling counters for the range defined. The scale for the **profil** call is calculated from the number of counters available for program address-sample counting and the address range defined by the *LowProgramCounter* and *HighProgramCounter* parameters. See the**mon.h** file.

Item	Description

NFunction	Defines the size of the space to be used for the function-count data area. The space is included as part of the first (or only) range buffer.
	
	When **-p** profiling is defined, the *NFunction* parameter defines the maximum number of functions to be counted. The space required for each function is defined to be:
	
	`sizeof(struct poutcnt)`
	
	The **poutcnt** structure is defined in the **mon.h** file. The total function-count space required is:
	
	`NFunction * sizeof(struct poutcnt)`
	
	When **-pg** profiling is defined, the *NFunction* parameter defines the size of the space (in bytes) available for the function-count data structures, as follows:
	
	`range = HighProgramCounter - LowProgramCounter; tonum = TO_NUM_ELEMENTS( range ); if ( tonum < MINARCS ) tonum = MINARCS; if ( tonum > TO_MAX-1 ) tonum = TO_MAX-1; tosize = tonum * sizeof( struct tostruct ); fromsize = FROM_STG_SIZE( range ); rangesize = tosize + fromsize + sizeof(struct gfctl);`
	
	This is computed and summed for all defined ranges. In this expression, the functions and variables in capital letters as well as the structures are defined in the **mon.h** file.

NotZeroA	Specifies a value of parameter 1, which is any value except 0. Ignored when it is not zero.
DoNotCareA	Specifies a value of parameter 2, of any value, which is ignored.

Return Values

The **monitor** subroutine returns 0 upon successful completion.

Error Codes

If an error is found, the **monitor** subroutine sends an error message to **stderr** and returns -1.

Examples

1. This example shows how to profile the main load module of a program with **-p** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern caddr_t etext;  /*system end of main module text symbol*/
extern int start();    /*first function in main program*/
extern struct monglobal _mondata; /*profiling global variables*/
struct desc {         /*function descriptor fields*/
    caddr_t begin;    /*initial code address*/
```

```
    caddr_t toc;      /*table of contents address*/
    caddr_t env;      /*environment pointer*/
} ;                  /*function descriptor structure*/
struct desc *fd;       /*pointer to function descriptor*/
int rc;               /*monitor return code*/
int range;             /*program address range for profiling*/
int numfunc;           /*number of functions*/
HISTCOUNTER *buffer;    /*buffer address*/
int numtics;        /*number of program address sample counters*/
int BufferSize;  /*total buffer size in numbers of HISTCOUNTERs*/
fd = (struct desc*)start; /*init descriptor pointer to start\
 function*/
numfunc = 300;          /*arbitrary number for example*/
range = etext - fd->begin; /*compute program address range*/
numtics =NUM_HIST_COUNTERS(range); /*one counter for each 4 byte\
 inst*/
BufferSize = numtics + ( numfunc*sizeof (struct poutcnt) \
 HIST_COUNTER_SIZE );     /*allocate buffer space*/
buffer = (HISTCOUNTER *) malloc (BufferSize * HIST_COUNTER_SIZE);
if ( buffer == NULL )  /*didn't get space, do error recovery\
 here*/
   return(-1);
_mondata.prof_type = _PROF_TYPE_IS_P; /*define -p profiling*/
rc = monitor( fd->begin, (caddr_t)etext, buffer, BufferSize, \
 numfunc);
/*start*/
if ( rc != 0 ) /*profiling did not start, do error recovery\
 here*/
   return(-1);
/*other code for analysis*/
rc = monitor( (caddr_t)0);  /*stop profiling and write data file\
 mon.out*/
if ( rc != 0 ) /*did not stop correctly, do error recovery here*/
   return (-1);
}
```

2. This example profiles the main program and the **libc.a** shared library with **-p** profiling. The range of addresses for the shared **libc.a** is assumed to be:

```
low = d0300000
high = d0312244
```

These two values can be determined from the **loadquery** subroutine at execution time, or by using a debugger to view the loaded programs' execution addresses and the loader map.

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern caddr_t etext;   /*system end of text symbol*/
extern int start();    /*first function in main program*/
extern struct monglobal _mondata; /*profiling global variables*/
struct prof pb[3];     /*prof array of 3 to define 2 ranges*/
int rc;               /*monitor return code*/
int range;            /*program address range for profiling*/
int numfunc;          /*number of functions to count (max)*/
int numtics;          /*number of sample counters*/
int num4fcnt;   /*number of HISTCOUNTERs used for fun cnt space*/
int BufferSize1;      /*first range BufferSize*/
int BufferSize2;      /*second range BufferSize*/
caddr_t liblo=0xd0300000;  /*lib low address (example only)*/
caddr_t libhi=0xd0312244;  /*lib high address (example only)*/
numfunc = 400;          /*arbitrary number for example*/
/*compute first range buffer size*/
range = etext - *(uint *) start;   /*init range*/
numtics = NUM_HIST_COUNTERS( range );
/*one counter for each 4 byte inst*/
num4fcnt = numfunc*sizeof( struct poutcnt )/HIST_COUNTER_SIZE;
BufferSize1 = numtics + num4fcnt;
/*compute second range buffer size*/
range = libhi-liblo;
BufferSize2 = range / 12; /*counter for every 12 inst bytes for\
 a change*/
/*allocate buffer space - note: must be single contiguous\
 buffer*/
pb[0].p_buff = (HISTCOUNTER *)malloc( (BufferSize1 +BufferSize2)\
 *HIST_COUNTER_SIZE);
if ( pb[0].p_buff == NULL )   /*didn't get space - do error\
```

```
 recovery here* ;/
    return(-1);
/*set up the first range values*/
pb[0].p_low = *(uint*)start;       /*start of main module*/
pb[0].p_high = (caddr_t)etext;     /*end of main module*/
pb[0].p_BufferSize = BufferSize1;  /*prog addr cnt space + \
func cnt space*/
/*set up last element marker*/
pb[2].p_high = (caddr_t)0;
_mondata.prof_type = _PROF_TYPE_IS_P;   /*define -p\
profiling*/
rc = monitor( (caddr_t)1, (caddr_t)1, pb, -1, numfunc); \
 /*start*/
if ( rc != 0 )   /*profiling did not start - do error recovery\
 here*/
   return (-1);
/*other code for analysis ...*/
rc = monitor( (caddr_t)0);   /*stop profiling and write data \
file mon.out*/
if ( rc != 0 )   /*did not stop correctly - do error recovery\
 here*/
    return (-1);
```

3. This example shows how to profile contiguously loaded functions beginning at `zit` up to but not including `zot` with **-pg** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern zit();           /*first function to profile*/
extern zot();           /*upper bound function*/
extern struct monglobal _mondata; /*profiling global variables*/
int rc;                /*monstartup return code*/
_mondata.prof_type = _PROF_TYPE_IS_PG; /*define -pg profiling*/
/*Note cast used to obtain function code addresses*/
rc = monstartup(*(uint *)zit,*(uint *)zot); /*start*/
if ( rc != 0 ) /*profiling did not start, do error recovery\
 here*/
   return(-1);
/*other code for analysis ...*/
exit(0);    /*stop profiling and write data file gmon.out*/
}
```

Files

Item	Description
mon.out	Data file for **-p** profiling.
gmon.out	Data file for **-pg** profiling.
/usr/include/mon.h	Defines the **_mondata.prof_type** global variable in the **monglobal** data structure, the **prof** structure, and the functions referred to in the previous examples.

monstartup Subroutine

Purpose

Starts and stops execution profiling using default-sized data areas.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mon.h>
```

int monstartup (*LowProgramCounter*, *HighProgramCounter*)

OR

int monstartup((caddr_t)-1), (caddr_t) *FragBuffer*)

OR

int monstartup((caddr_t)-1, (caddr_t)0)

```
caddr_t LowProgramCounter;
caddr_t HighProgramCounter;
```

Description

The **monstartup** subroutine allocates data areas of default size and starts profiling. Profiling causes periodic sampling and recording of the program location within the program address ranges specified, and accumulation of function-call count data for functions that have been compiled with the **-p** or **-pg** option.

Executable programs created with the **cc -p** or **cc -pg** command automatically include a call to the **monstartup** subroutine to profile the complete user program, including system libraries. In this case, you do not need to call the **monstartup** subroutine.

The **monstartup** subroutine is called by the **mcrt0.o** (**-p**) file or the **gcrt0.o** (**-pg**) file to begin profiling. The **monstartup** subroutine requires a global data variable to define whether **-p** or **-pg** profiling is to be in effect. The **monstartup** subroutine calls the **monitor** subroutine to initialize the data areas and start profiling.

The **prof** command is used to process the data file produced by **-p** profiling. The **gprof** command is used to process the data file produced by **-pg** profiling.

The **monstartup** subroutine examines the global and parameter data in the following order:

1. When the **_mondata.prof_type** global variable is neither -1 (**-p** profiling defined) nor +1 (**-pg** profiling defined), an error is returned and the function is considered complete.

   The global variable is set to -1 in the **mcrt0.o** file and to +1 in the **gcrt0.o** file, and defaults to 0 when **crt0.o** is used.
2. When the *LowProgramCounter* value is not -1:

   - A single program address range is defined for profiling

   AND

   - The first **monstartup** definition in the syntax is used to define the program range.
3. When the *LowProgramCounter* value is -1 and the *HighProgramCounter* value is not 0:

   - Multiple program address ranges are defined for profiling

   AND

   - The second **monstartup** definition in the syntax is used to define multiple ranges. The *HighProgramCounter* parameter, in this case, is the address of a **frag** structure array. The **frag** array size is denoted by a zero value for the *HighProgramCounter* (p_high) field of the last element of the array. Each array element except the last defines one programming address range to be profiled. Programming ranges must be in ascending order of the program addresses with ascending order of the **prof** array index. Program ranges may not overlap.
4. When the *LowProgramCounter* value is -1 and the *HighProgramCounter* value is 0:

- The whole program is defined for profiling

AND

- The third **monstartup** definition in the syntax is used. The program ranges are determined by **monstartup** and may be single range or multirange.

Parameters

Item	Description
LowProgramCounter (**frag** name: p_low)	Defines the lowest execution-time program address in the range to be profiled.
HighProgramCounter(**frag** name: p_high)	Defines the next address after the highest execution-time program address in the range to be profiled. The program address parameters may be defined by function names or address expressions. If defined by a function name, then a function name expression must be used to dereference the function pointer to get the address of the first instruction in the function. This is required because the function reference in this context produces the address of the function descriptor. The first field of the descriptor is the address of the function code. See the examples for typical expressions to use.
FragBuffer	Specifies the address of a frag structure array.

Examples

1. This example shows how to profile the main load module of a program with **-p** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern caddr_t etext;      /*system end of text
symbol*/
extern int start();      /*first function in main\
            program*/
extern struct monglobal _mondata; /*profiling global variables*/
struct desc {         /*function
descriptor fields*/
   caddr_t begin;      /*initial code
address*/
   caddr_t toc;      /*table of contents
address*/
   caddr_t env;      /*environment
pointer*/
 }
;            /*function
descriptor structure*/
struct desc *fd;      /*pointer to function\
            descriptor*/
int  rc;         /*monstartup
return code*/
fd = (struct desc *)start;   /*init descriptor pointer to\
            start
function*/
_mondata.prof_type = _PROF_TYPE_IS_P;   /*define -p profiling*/
rc = monstartup( fd->begin, (caddr_t) &etext);   /*start*/
if ( rc != 0 )         /*profiling did
not start - do\
           error
recovery here*/   return(-1);
           /*other code
for analysis ...*/
```

```
return(0);         /*stop profiling and
write data\
             file
mon.out*/
}
```

2. This example shows how to profile the complete program with **-p** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern struct monglobal _mondata;   /*profiling global\
                variables*/
int  rc;         /*monstartup
return code*/
_mondata.prof_type = _PROF_TYPE_IS_P;   /*define -p profiling*/
rc = monstartup( (caddr_t)-1, (caddr_t)0);   /*start*/
if ( rc != 0 )         /*profiling did
not start -\

  do error recovery here*/
   return (-1);
            /*other code
for analysis ...*/
return(0);         /*stop profiling and
write data\
             file
mon.out*/
}
```

3. This example shows how to profile contiguously loaded functions beginning at `zit` up to but not including `zot` with **-pg** profiling:

```
#include <sys/types.h>
#include <mon.h>
main()
{
extern zit();         /*first function
to profile*/
extern zot();         /*upper bound
function*/
extern struct monglobal _mondata; /*profiling global variables*/
int  rc;         /*monstartup
return code*/
_mondata.prof_type = _PROF_TYPE_IS_PG;   /*define -pg profiling*/
/*Note cast used to obtain function code addresses*/
rc = monstartup(*(uint *)zit,*(uint *)zot);   /*start*/
if ( rc != 0 )         /*profiling did
not start - do\
             error
recovery here*/
   return(-1);
            /*other code
for analysis ...*/
exit(0);   /*stop profiling and write data file gmon.out*/
}
```

Return Values

The **monstartup** subroutine returns 0 upon successful completion.

Error Codes

If an error is found, the **monstartup** subroutine outputs an error message to **stderr** and returns -1.

Files

Item	Description
mon.out	Data file for **-p** profiling.
gmon.out	Data file for **-pg** profiling.

Item	Description
mon.h	Defines the **_mondata.prof_type** variable in the **monglobal** data structure, the **prof** structure, and the functions referred to in the examples.

move or wmove Subroutine

Purpose

Window location cursor functions.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int move (int y, int x);
int wmove (WINDOW *win, int y, int x);
```

Description

The **move** and **wmove** subroutines move the logical cursor associated with the current or specified window to (y, x) relative to the window's origin. This subroutine does not move the cursor of the terminal until the next **refresh** (“refresh or wrefresh Subroutine” on page 1728) operation.

Parameters

Item	Description
y	Holds the line or row coordinate of the logical cursor.
x	Holds the column coordinate of the logical cursor.
**win*	Identifies the window in which the cursor is being moved.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To move the logical cursor in the stdscr to the coordinates y = 5, x = 10, use:

   ```
   move(5, 10);
   ```

2. To move the logical cursor in the user-defined window `my_window` to the coordinates y = 5, x = 10, use:

   ```
   WINDOW *my_window;
   wmove(my_window, 5, 10);
   ```

mprotect Subroutine

Purpose

Modifies access protections for memory mapping or shared memory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>
```

```
int mprotect ( addr,  len,  prot)
void *addr;
size_t len;
int prot;
```

Description

The **mprotect** subroutine modifies the access protection of a mapped file or shared memory region or anonymous memory region created by the **mmap** subroutine. Processes running in an environment where the MPROTECT_SHM=ON environmental variable is defined can also use the **mprotect** subroutine to modify the access protection of a shared memory region created by the **shmget**, **ra_shmget**, or **ra_shmgetv** subroutine and attached by the **shmat** subroutine.

Processes running in an environment where the MPROTECT_TXT=ON environmental variable is defined can use the **mprotect** subroutine to modify access protections on main text, shared library, and loaded code. There is no requirement for these areas to be mapped using the **mmap** subroutine prior to their modification by the **mprotect** subroutine. A private copy of any modification to the application text is made using the copy-on-write semantics. Modifications to the content of application text are not persistent. Modifications to the application text will be propagated to the child processes across fork calls. Subsequent modifications by forker and sibling remain private to each other.

The user who protects shared memory with the **mprotect** subroutine must be also be either the user who created the shared memory descriptor, the user who owns the shared memory descriptor, or the root user.

The **mprotect** subroutine can only be used on shared memory regions backed with 4 KB or 64 KB pages; shared memory regions backed by 16 MB and 16 GB pages are not supported by the **mprotect** subroutine. The page size used to back a shared memory region can be obtained using the **vmgetinfo** subroutine and specifying VM_PAGE_INFO for the *command* parameter.

The **mprotect** subroutine cannot be used for shared memory that has been pre-translated. This includes shared memory regions created with the SHM_PIN flag specified to the **shmget** subroutine as well as shared memory regions that have been pinned using the **shmctl** subroutine with the SHM_LOCK flag specified.

Parameters

addr
Specifies the address of the region to be modified. Must be a multiple of the page size backing the memory region.

len
Specifies the length, in bytes, of the region to be modified. For shared memory regions backed with 4 KB pages, the *len* parameter will be rounded off to the next multiple of the page size. Otherwise, the *len* parameter must be a multiple of the page size backing the memory region.

prot
Specifies the new access permissions for the mapped region. Legitimate values for the *prot* parameter are the same as those permitted for the **mmap** subroutine, as follows:

PROT_READ
Region can be read.

PROT_WRITE
Region can be written.

PROT_EXEC
Region can be executed.

PROT_NONE
Region cannot be accessed. PROT_NONE is not a valid *prot* parameter for shared memory attached with the **shmat** subroutine.

Return Values

When successful, the **mprotect** subroutine returns 0. Otherwise, it returns -1 and sets the **errno** global variable to indicate the error.

Note: The return value for the **mprotect** subroutine is 0 if it fails because the region given was not created by **mmap** unless XPG 1170 behavior is requested by setting the **XPG_SUS_ENV** environment variable to **ON**.

Error Codes

If the **mprotect** subroutine is unsuccessful, the **errno** global variable might be set to one of the following values:

Attention: If the **mprotect** subroutine is unsuccessful because of a condition other than that specified by the **EINVAL** error code, the access protection for some pages in the (*addr*, *addr* + *len*) range might have been changed.

Item	Description
EACCES	The *prot* parameter specifies a protection that conflicts with the access permission set for the underlying file.
EPERM	The user is not the creator or owner of the shared memory region and is not the root user.
ENOTSUP	The *prot* parameter specified is not valid for the region specified.
EINVAL	The *addr* or *len* parameter is not a multiple of the page size backing the memory region.
ENOMEM	The application has requested Single UNIX Specification, Version 2 compliant behavior, but addresses in the range are not valid for the address space of the process, or the addresses specify one or more pages that are not attached to the user's address space by a previous **mmap** or **shmat** subroutine call.
ENOTSUP	The shared memory region specified is backed by 64 KB pages, but the *addr* or *len* parameter is not 64 KB aligned, or PROT_NONE protection was specified for a shared memory region, or a pre-translated shared memory region was specified, or a shared memory region backed by 16 MB or 16 GB pages was specified.

mq_close Subroutine

Purpose

Closes a message queue.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mqueue.h>

int mq_close (mqdes)
mqd_t mqdes;
```

Description

The **mq_close** subroutine removes the association between the message queue descriptor, *mqdes*, and its message queue. The results of using this message queue descriptor after successful return from the **mq_close** subroutine, and until the return of this message queue descriptor from a subsequent **mq_open** call, are undefined.

If the process has successfully attached a notification request to the message queue through the *mqdes* parameter, this attachment is removed, and the message queue is available for another process to attach for notification.

Parameters

Item	Description
mqdes	Specifies the message queue descriptor.

Return Values

Upon successful completion, the **mq_close** subroutine returns a zero. Otherwise, the subroutine returns a -1 and sets **errno** to indicate the error.

Error Codes

The **mq_close** subroutine fails if:

Item	Description
EBADF	The *mqdes* parameter is not a valid message queue descriptor.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

mq_getattr Subroutine

Purpose

Gets message queue attributes.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mqueue.h>
```

```
int mq_getattr (mqdes, mqstat)
mqd_t mqdes;
struct mq_attr *mqstat;
```

Description

The **mq_getattr** subroutine obtains status information and attributes of the message queue and the open message queue description associated with the message queue descriptor.

The results are returned in the **mq_attr** structure referenced by the *mqstat* parameter.

Upon return, the following members have the values associated with the open message queue description as set when the message queue was opened and as modified by subsequent calls to the **mq_setattr** subroutine:

- *mq_flags*

The following attributes of the message queue are returned as set at message queue creation:

- *mq_maxmsg*
- *mq_msgsize*

Upon return, the following member within the **mq_attr** structure referenced by the *mqstat* parameter is set to the current state of the message queue:

Item	Description
mq_curmsgs	The number of messages currently on the queue.

Parameters

Item	Description
mqdes	Specifies a message queue descriptor.
mqstat	Points to the **mq_attr** structure.

Return Values

Upon successful completion, the **mq_getattr** subroutine returns zero. Otherwise, the subroutine returns -1 and sets **errno** to indicate the error.

Error Codes

The **mq_getattr** subroutine fails if:

Item	Description
EBADF	The *mqdes* parameter is not a valid message queue descriptor.
EFAULT	Invalid user address.
EINVAL	The *mqstat* parameter value is not valid.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

mq_notify Subroutine

Purpose

Notifies a process that a message is available.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mqueue.h>

int mq_notify (mqdes, notification)
mqd_t mqdes;
const struct sigevent *notification;
```

Description

If the *notification* parameter is not NULL, the **mq_notify** subroutine registers the calling process to be notified of message arrival at an empty message queue associated with the specified message queue descriptor, *mqdes*. The notification specified by the *notification* parameter is sent to the process when the message queue transitions from empty to non-empty. At any time only one process may be registered for notification by a message queue. If the calling process or any other process has already registered for notification of message arrival at the specified message queue, subsequent attempts to register for that message queue fails.

If notification is NULL and the process is currently registered for notification by the specified message queue, the existing registration is removed.

When the notification is sent to the registered process, its registration is removed. The message queue is then available for registration.

If a process has registered for notification of message arrival at a message queue and a thread is blocked in the **mq_receive** or **mq_timedreceive** subroutines waiting to receive a message, the arriving message satisfies the appropriate **mq_receive** or **mq_timedreceive** subroutine respectively. The resulting behavior is as if the message queue remains empty, and no notification is sent.

Parameters

Item	Description
mqdes	Specifies a message queue descriptor.
notification	Points to the **sigevent** structure.

Return Values

Upon successful completion, the **mq_notify** subroutine returns a zero. Otherwise, it returns a value of -1 and sets **errno** to indicate the error.

Error Codes

The **mq_notify** subroutine fails if:

Item	Description
EBADF	The *mqdes* parameter is not a valid message queue descriptor.
EBUSY	A process is already registered for notification by the message queue.
EFAULT	Invalid used address.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.
EINVAL	The current process is not registered for notification for the specified message queue and registration removal was requested.

mq_open Subroutine

Purpose

Opens a message queue.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mqueue.h>

mqd_t mq_open (name, oflag [mode, attr])
const char *name;
int oflag;
mode_t mode;
mq_attr *attr;
```

Description

The **mq_open** subroutine establishes a connection between a process and a message queue with a message queue descriptor. It creates an open message queue description that refers to the message queue, and a message queue descriptor that refers to that open message queue description. The message queue descriptor is used by other subroutines to refer to that message queue.

The *name* parameter points to a string naming a message queue, and has no representation in the file system. The *name* parameter conforms to the construction rules for a pathname. It may or may not begin with a slash character, but contains at least one character. Processes calling the **mq_open** subroutine with the same value of *name* refer to the same message queue object, as long as that name has not been removed. If the *name* parameter is not the name of an existing message queue and creation is not requested, the **mq_open** subroutine will fail and return an error.

The *oflag* parameter requests the desired receive and send access to the message queue. The requested access permission to receive messages or send messages is granted if the calling process would be granted read or write access, respectively, to an equivalently protected file.

The value of the *oflag* parameter is the bitwise-inclusive OR of values from the following list. Applications specify exactly one of the first three values (access modes) below in the value of the *oflag* parameter:

O_RDONLY
Open the message queue for receiving messages. The process can use the returned message queue descriptor with the **mq_receive** subroutine, but not the **mq_send** subroutine. A message queue may be open multiple times in the same or different processes for receiving messages.

O_WRONLY
Open the queue for sending messages. The process can use the returned message queue descriptor with the **mq_send** subroutine but not the **mq_receive** subroutine. A message queue may be open multiple times in the same or different processes for sending messages.

O_RDWR
Open the queue for both receiving and sending messages. The process can use any of the functions allowed for the **O_RDONLY** and **O_WRONLY** flags. A message queue may be open multiple times in the same or different processes for sending messages.

Any combination of the remaining flags may be specified in the value of the *oflag* parameter:

O_CREAT
Create a message queue. It requires two additional arguments: *mode*, which is of **mode_t** type, and *attr*, which is a pointer to an **mq_attr** structure. If the pathname *name* has already been used to create a message queue that still exists, this flag has no effect, except as noted under the **O_EXCL** flag. Otherwise, a message queue is created without any messages in it. The user ID of the message

queue is set to the effective user ID of the process, and the group ID of the message queue is set to the effective group ID of the process. The file permission bits are set to the value of *mode*. When bits in the *mode* parameter other than file permission bits are set, they have no effect. If *attr* is NULL, the message queue is created with default message queue attributes. Default values are 128 for *mq_maxmsg* and 1024 for *mq_msgsize*. If *attr* is non-NULL, the message queue *mq_maxmsg* and *mq_msgsize* attributes are set to the values of the corresponding members in the **mq_attr** structure referred to by *attr*.

O_EXCL
If the **O_EXCL** and **O_CREAT** flags are set, the **mq_open** subroutine fails if the message queue name exists. The check for the existence of the message queue and the creation of the message queue if it does not exist is atomic with respect to other threads executing **mq_open** naming the same name with the **O_EXCL** and **O_CREAT** flags set. If the **O_EXCL** flag is set and the **O_CREAT** flag is not set, the **O_EXCL** flag is ignored.

O_NONBLOCK
Determines whether the **mq_send** or **mq_receive** subroutine waits for resources or messages that are not currently available, or fails with **errno** set to **EAGAIN**; see **mq_send** and **mq_receive** for details.

The **mq_open** subroutine does not add or remove messages from the queue.

Parameters

Item	Description
name	Points to a string naming a message queue.
oflag	Requests the desired receive and send access to the message queue.
mode	Specifies the value of the file permission bits. Used with **O_CREAT** to create a message queue.
attr	Points to an **mq_attr** structure. Used with **O_CREAT** to create a message queue.

Return Values

Upon successful completion, the **mq_open** subroutine returns a message queue descriptor. Otherwise, it returns (**mqd_t**)-1 and sets **errno** to indicate the error.

Error Codes

The **mq_open** subroutine fails if:

Item	Description
EACCES	The message queue exists and the permissions specified by the *oflag* parameter are denied.
EEXIST	The **O_CREAT** and **O_EXCL** flags are set and the named message queue already exists.
EFAULT	Invalid used address.
EINVAL	The **mq_open** subroutine is not supported for the given name.
EINVAL	The **O_CREAT** flag was specified in the *oflag* parameter, the value of *attr* is not NULL, and either *mq_maxmsg* or *mq_msgsize* was less than or equal to zero.
EINVAL	The *oflag* parameter value is not valid.
EMFILE	Too many message queue descriptors are currently in use by this process.
ENAMETOOLONG	The length of the *name* parameter exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.

Item	Description
ENFILE	Too many message queues are currently open in the system.
ENOENT	The **O_CREAT** flag is not set and the named message queue does not exist.
ENOMEM	Insufficient memory for the required operation.
ENOSPC	There is insufficient space for the creation of the new message queue.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

mq_receive Subroutine

Purpose

Receives a message from a message queue.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mqueue.h>

ssize_t mq_receive (mqdes, msg_ptr, msg_len, msg_prio)
mqd_t mqdes;
char *msg_ptr;
size_t msg_len;
unsigned *msg_prio;
```

Description

The **mq_receive** subroutine receives the oldest of the highest priority messages from the message queue specified by the *mqdes* parameter. If the size of the buffer in bytes, specified by the *msg_len* parameter, is less than the *mq_msgsize* attribute of the message queue, the subroutine fails and returns an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by the *msg_ptr* parameter.

If the *msg_prio* parameter is not NULL, the priority of the selected message is stored in the location referenced by *msg_prio*.

If the specified message queue is empty and the **O_NONBLOCK** flag is not set in the message queue description associated with the *mqdes* parameter, the **mq_receive** subroutine blocks until a message is enqueued on the message queue or until **mq_receive** is interrupted by a signal. If more than one thread is waiting to receive a message when a message arrives at an empty queue and the Priority Scheduling option is supported, the thread of highest priority that has been waiting the longest is selected to receive the message. If the specified message queue is empty and the **O_NONBLOCK** flag is set in the message queue description associated with the *mqdes* parameter, no message is removed from the queue, and the **mq_receive** subroutine returns an error.

Parameters

Item	Description
mqdes	Specifies the message queue descriptor.
msg_ptr	Points to the buffer where the message is copied.
msg_len	Specifies the length of the message, in bytes.

Item	Description
msg_prio	Stores the priority of the selected message.

Return Values

Upon successful completion, the **mq_receive** subroutine returns the length of the selected message in bytes and the message is removed from the queue. Otherwise, no message is removed from the queue, and the subroutine returns -1 and sets **errno** to indicate the error.

Error Codes

The **mq_receive** subroutine fails if:

Item	Description
EAGAIN	The **O_NONBLOCK** flag was set in the message description associated with the *mqdes* parameter, and the specified message queue is empty.
EBADF	The *mqdes* parameter is not a valid message queue descriptor open for reading.
EFAULT	Invalid used address.
EIDRM	The specified message queue was removed during the required operation.
EINTR	The **mq_receive** subroutine was interrupted by a signal.
EINVAL	The *msg_ptr* parameter is null.
EMSGSIZE	The specified message buffer size, *msg_len*, is less than the message size attribute of the message queue.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

mq_send Subroutine

Purpose

Sends a message to a message queue.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mqueue.h>

int mq_send (mqdes, msg_ptr, msg_len, msg_prio)
mqd_t mqdes;
const char *msg_ptr;
size_t msg_len;
unsigned *msg_prio;
```

Description

The **mq_send** subroutine adds the message pointed to by the *msg_ptr* parameter to the message queue specified by the *mqdes* parameter. The *msg_len* parameter specifies the length of the message, in bytes, pointed to by *msg_ptr*. The value of *msg_len* is less than or equal to the *mq_msgsize* attribute of the message queue, or the **mq_send** subroutine will fail.

If the specified message queue is not full, the **mq_send** subroutine behaves as if the message is inserted into the message queue at the position indicated by the *msg_prio* parameter. A message with a larger numeric value of *msg_prio* will be inserted before messages with lower values of *msg_prio*. A message will be inserted after other messages in the queue with equal *msg_prio*. The value of *msg_prio* will be less than **MQ_PRIO_MAX**.

If the specified message queue is full and **O_NONBLOCK** is not set in the message queue description associated with *mqdes*, the **mq_send** subroutine will block until space becomes available to enqueue the message, or until **mq_send** is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue and the Priority Scheduling option is supported, the thread of the highest priority that has been waiting the longest is unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and **O_NONBLOCK** is set in the message queue description associated with *mqdes*, the message is not queued and the **mq_send** subroutine returns an error.

Parameters

Item	Description
mqdes	Specifies the message queue descriptor.
msg_ptr	Points to the message to be added.
msg_len	Specifies the length of the message, in bytes.
msg_prio	Specifies the position of the message in the message queue.

Return Values

Upon successful completion, the **mq_send** subroutine returns a zero. Otherwise, no message is enqueued, the subroutine returns -1, and **errno** is set to indicate the error.

Error Codes

The **mq_send** subroutine fails if:

Item	Description
EAGAIN	The **O_NONBLOCK** flag is set in the message queue description associated with the *mqdes* parameter, and the specified message queue is full (maximum number of messages in the queue or maximum number of bytes in the queue is reached).
EBADF	The *mqdes* parameter is not a valid message queue descriptor open for writing.
EFAULT	Invalid used address.
EIDRM	The specified message queue was removed during the required operation.
EINTR	A signal interrupted the call to the **mq_send** subroutine.
EINVAL	The value of the *msg_prio* parameter was outside the valid range.
EINVAL	The *msg_ptr* parameter is null.
EMSGSIZE	The specified message length, *msg_len*, exceeds the message size attribute of the message queue.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

mq_setattr Subroutine

Purpose

Sets message queue attributes.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mqueue.h>

int mq_setattr (mqdes, mqstat, omqstat)
mqd_t mqdes;
const struct mq_attr *mqstat;
struct mq_attr *omqstat;
```

Description

The **mq_setattr** subroutine sets attributes associated with the open message queue description referenced by the message queue descriptor specified by *mqdes*.

The message queue attributes corresponding to the following members defined in the **mq_attr** structure are set to the specified values upon successful completion of the **mq_setattr** subroutine.

The value of the *mq_flags* member is either zero or **O_NONBLOCK**.

The values of the *mq_maxmsg*, *mq_msgsize*, and *mq_curmsgs* members of the **mq_attr** structure are ignored by the **mq_setattr** subroutine.

If the *omqstat* parameter is non-NULL, the **mq_setattr** subroutine stores, in the location referenced by *omqstat*, the previous message queue attributes and the current queue status. These values are the same as would be returned by a call to the **mq_getattr** subroutine at that point.

Parameters

Item	Description
mqdes	Specifies the message queue descriptor.
mqstat	Specifies the status of the message queue.
omqstat	Specifies the status of the previous message queue.

Return Values

Upon successful completion, the **mq_setattr** subroutine returns a zero and the attributes of the message queue are changed as specified.

Otherwise, the message queue attributes are unchanged, and the subroutine returns a -1 and sets **errno** to indicate the error.

Error Codes

The **mq_setattr** subroutine fails if:

Item	Description
EBADF	The *mqdes* parameter is not a valid message queue descriptor.
EFAULT	Invalid user address.

Item	Description
EINVAL	The *mqstat* parameter value is not valid.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

mq_receive, mq_timedreceive Subroutine

Purpose

Receives a message from a message queue (REALTIME).

Syntax

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
       size_t msg_len, unsigned *msg_prio,


#include <mqueue.h>
#include <time.h>

ssize_t mq_timedreceive(mqd_t mqdes, char *restrict msg_ptr,
       size_t msg_len, unsigned *restrict msg_prio,
       const struct timespec *restrict abs_timeout);
```

Description

The **mq_receive()** function receives the oldest of the highest priority messages from the message queue specified by *mqdes*. If the size of the buffer, in bytes, specified by the *msg_len* argument is less than the *mq_msgsize* attribute of the message queue, the function fails and returns an error. Otherwise, the selected message is removed from the queue and copied to the buffer pointed to by the *msg_ptr* argument.

If the value of *msg_len* is greater than {SSIZE_MAX}, the result is implementation-defined.

If the *msg_prio* argument is not NULL, the priority of the selected message is stored in the location referenced by *msg_prio*.

If the specified message queue is empty and O_NONBLOCK is not set in the message queue description associated with *mqdes*, **mq_receive()** blocks until a message is enqueued on the message queue or until **mq_receive()** is interrupted by a signal. If more than one thread is waiting to receive a message when a message arrives at an empty queue and the Priority Scheduling option is supported, then the thread of highest priority that has been waiting the longest is selected to receive the message. Otherwise, it is unspecified which waiting thread receives the message. If the specified message queue is empty and O_NONBLOCK is set in the message queue description associated with *mqdes*, no message is removed from the queue, and **mq_receive()** returns an error.

The **mq_timedreceive()** function receives the oldest of the highest priority messages from the message queue specified by *mqdes* as described for the **mq_receive()** function. However, if O_NONBLOCK was not specified when the message queue was opened by the **mq_open()** function, and no message exists on the queue to satisfy the receive, the wait for such a message is terminated when the specified timeout expires. If O_NONBLOCK is set, this function matches **mq_receive()**.

The timeout expires when the absolute time specified by *abs_timeout* passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*), or when the absolute time specified by *abs_timeout* has already been passed at the time of the call.

If the **Timers** option is supported, the timeout is based on the CLOCK_REALTIME clock; if the **Timers** option is not supported, the timeout is based on the system clock as returned by the **time()** function.

The resolution of the timeout matches the resolution of the clock on which it is based. The *timespec* argument is defined in the **<time.h>** header.

The operation never fails with a timeout if a message can be removed from the message queue immediately. The validity of the *abs_timeout* parameter does not need to be checked if a message can be removed from the message queue immediately.

Return Values

Upon successful completion, the **mq_receive()** and **mq_timedreceive()** functions return the length of the selected message in bytes and the message is removed from the queue. Otherwise, no message shall be removed from the queue, the functions return a value of -1, and *errno* is set to indicate the error.

Error Codes

The **mq_receive()** and **mq_timedreceive()** functions fail if:

Item	Description
[EAGAIN]	O_NONBLOCK was set in the message description associated with *mqdes*, and the specified message queue is empty.
[EBADF]	The *mqdes* argument is not a valid message queue descriptor open for reading.
[EFAULT]	*abs_timeout* references invalid memory.
[EIDRM]	Specified message queue was removed during required operation.
[EINTR]	The **mq_receive()** or **mq_timedreceive()** operation was interrupted by a signal.
[EINVAL]	The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.
[EINVAL]	*msg_ptr* value was null.
[EMSGSIZE]	The specified message buffer size, *msg_len*, is less than the message size attribute of the message queue.
[ENOTSUP]	Function is not supported with checkpoint-restart'ed processes.
[ETIMEDOUT]	The O_NONBLOCK flag was not set when the message queue was opened, but no message arrived on the queue before the specified timeout expired.

The **mq_receive()** and **mq_timedreceive()** functions might fail if:

Item	Description
[EBADMSG]	The implementation has detected a data corruption problem with the message.

mq_send, mq_timedsend Subroutine

Purpose

Sends a message to a message queue (REALTIME).

Syntax

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *msg_ptr,
```

```
        size_t msg_len, unsigned *msg_prio,


#include <mqueue.h>
#include <time.h>

int mq_timedsend(mqd_t mqdes, const char *msg_ptr,
        size_t msg_len, unsigned msg_prio,
        const struct timespec *abs_timeout);
```

Description

The **mq_send()** function adds the message pointed to by the argument *msg_ptr* to the message queue specified by *mqdes*. The *msg_len* argument specifies the length of the message, in bytes, pointed to by *msg_ptr*. The value of *msg_len* is less than or equal to the *mq_msgsize* attribute of the message queue, or **mq_send()** fails.

If the specified message queue is not full, **mq_send()** behaves as if the message is inserted into the message queue at the position indicated by the *msg_prio* argument. A message with a larger numeric value of *msg_prio* is inserted before messages with lower values of *msg_prio*. A message is inserted after other messages in the queue, if any, with equal *msg_prio* values. The value of *msg_prio* is less than {MQ_PRIO_MAX}.

If the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, **mq_send()** blocks until space becomes available to enqueue the message, or until **mq_send()** is interrupted by a signal. If more than one thread is waiting to send when space becomes available in the message queue and the **Priority Scheduling** option is supported, then the thread of the highest priority that has been waiting the longest is unblocked to send its message. Otherwise, it is unspecified which waiting thread is unblocked. If the specified message queue is full and O_NONBLOCK is set in the message queue description associated with *mqdes*, the message is not queued and **mq_send()** returns an error.

The **mq_timedsend()** function adds a message to the message queue specified by *mqdes* in the manner defined for the **mq_send()** function. However, if the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue is terminated when the specified timeout expires. If O_NONBLOCK is set in the message queue description, this function matches **mq_send()**.

The timeout expires when the absolute time specified by *abs_timeout* passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds abs_timeout)—or when the absolute time specified by *abs_timeout* has already been passed at the time of the call.

If the **Timers** option is supported, the timeout is based on the CLOCK_REALTIME clock; if the **Timers** option is not supported, the timeout is based on the system clock as returned by the **time()** function.

The operation never fails with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the *abs_timeout* parameter does not need to be checked when there is sufficient room in the queue.

Application Usage

The value of the symbol {MQ_PRIO_MAX} limits the number of priority levels supported by the application. Message priorities range from 0 to {MQ_PRIO_MAX}-1.

Return Values

Upon successful completion, the **mq_send()** and **mq_timedsend()** functions return a value of 0. Otherwise, no message is enqueued, the functions return -1, and *errno* is set to indicate the error.

Error Codes

The **mq_send()** and **mq_timedsend()** functions fail if:

Item	Description
[EAGAIN]	The O_NONBLOCK flag is set in the message queue description associated with *mqdes*, and the specified message queue is full.
[EBADF]	The *mqdes* argument is not a valid message queue descriptor open for writing.
[EFAULT]	*abs_timeout* references invalid memory.
[EIDRM]	Specified message queue was removed during required operation.
[EINTR]	A signal interrupted the call to **mq_send()** or **mq_timedsend()**.
[EINVAL]	The value of *msg_prio* was outside the valid range.
[EINVAL]	*msg_ptr* value was null.
[EINVAL]	The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.
[EMSGSIZE]	The specified message length, *msg_len*, exceeds the message size attribute of the message queue.
[ENOTSUP]	Function is not supported with checkpoint-restart'ed processes.
[ETIMEDOUT]	The O_NONBLOCK flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

The **mq_send()** and **mq_timedsend()** functions might fail if:

Item	Description
[EBADMSG]	The implementation has detected a data corruption problem with the message.

mq_unlink Subroutine

Purpose

Removes a message queue.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <mqueue.h>

int mq_unlink (name)
const char *name;
```

Description

The **mq_unlink** subroutine removes the message queue named by the pathname *name*. After a successful call to the **mq_unlink** subroutine with the *name* parameter, a call to the **mq_open** subroutine with the *name* parameter and the **O_CREAT** flag will create a new message queue. If one or more processes have the message queue open when the **mq_unlink** subroutine is called, destruction of the message queue is postponed until all references to the message queue have been closed.

After a successful completion of the **mq_unlink** subroutine, calls to the **mq_open** subroutine to recreate a message queue with the same name will succeed. The **mq_unlink** subroutine never blocks even if all references to the message queue have not been closed.

Parameters

Item	Description
name	Specifies the message queue to be removed.

Return Values

Upon successful completion, the **mq_unlink** subroutine returns a zero. Otherwise, the named message queue is unchanged, and the **mq_unlink** subroutine returns a -1 and sets **errno** to indicate the error.

Error Codes

The **mq_unlink** subroutine fails if:

Item	Description
EACCES	Permission is denied to unlink the named message queue.
EFAULT	Invalid used address.
EINVAL	The *name* parameter value is not valid
ENAMETOOLONG	The length of the *name* parameter exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.
ENOENT	The named message queue does not exist.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

msem_init Subroutine

Purpose

Initializes a semaphore in a mapped file or shared memory region.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>
```

```
msemaphore *msem_init ( Sem,  InitialValue)
msemaphore *Sem;
int InitialValue;
```

Description

The **msem_init** subroutine allocates a new binary semaphore and initializes the state of the new semaphore.

If the value of the *InitialValue* parameter is **MSEM_LOCKED**, the new semaphore is initialized in the locked state. If the value of the *InitialValue* parameter is **MSEM_UNLOCKED**, the new semaphore is initialized in the unlocked state.

The **msemaphore** structure is located within a mapped file or shared memory region created by a successful call to the **mmap** subroutine and having both read and write access.

Whether a semaphore is created in a mapped file or in an anonymous shared memory region, any reference by a process that has mapped the same file or shared region, using an **msemaphore** structure pointer that resolved to the same file or start of region offset, is taken as a reference to the same semaphore.

Any previous semaphore state stored in the **msemaphore** structure is ignored and overwritten.

Parameters

Item	Description
Sem	Points to an **msemaphore** structure in which the state of the semaphore is stored.
Initial Value	Determines whether the semaphore is locked or unlocked at allocation.

Return Values

When successful, the **msem_init** subroutine returns a pointer to the initialized **msemaphore** structure. Otherwise, it returns a null value and sets the **errno** global variable to indicate the error.

Error Codes

If the **msem_init** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

Item	Description
EINVAL	Indicates the *InitialValue* parameter is not valid.
ENOMEM	Indicates a new semaphore could not be created.

msem_lock Subroutine

Purpose

Locks a semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>
```

```
int msem_lock ( Sem,  Condition)
msemaphore *Sem;
int Condition;
```

Description

The **msem_lock** subroutine attempts to lock a binary semaphore.

If the semaphore is not currently locked, it is locked and the **msem_lock** subroutine completes successfully.

If the semaphore is currently locked, and the value of the *Condition* parameter is **MSEM_IF_NOWAIT**, the **msem_lock** subroutine returns with an error. If the semaphore is currently locked, and the value of the *Condition* parameter is 0, the **msem_lock** subroutine does not return until either the calling process is able to successfully lock the semaphore or an error condition occurs.

All calls to the **msem_lock** and **msem_unlock** subroutines by multiple processes sharing a common **msemaphore** structure behave as if the call were serialized.

If the **msemaphore** structure contains any value not resulting from a call to the **msem_init** subroutine, followed by a (possibly empty) sequence of calls to the **msem_lock** and **msem_unlock** subroutines, the results are undefined. The address of an **msemaphore** structure is significant. If the **msemaphore** structure contains any value copied from an **msemaphore** structure at a different address, the result is undefined.

Parameters

Item	Description
Sem	Points to an **msemaphore** structure that specifies the semaphore to be locked.
Condition	Determines whether the **msem_lock** subroutine waits for a currently locked semaphore to unlock.

Return Values

When successful, the **msem_lock** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **msem_lock** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

Item	Description
EAGAIN	Indicates a value of **MSEM_IF_NOWAIT** is specified for the *Condition* parameter and the semaphore is already locked.
EINVAL	Indicates the *Sem* parameter points to an **msemaphore** structure specifying a semaphore that has been removed, or the *Condition* parameter is invalid.
EINTR	Indicates the **msem_lock** subroutine was interrupted by a signal that was caught.

msem_remove Subroutine

Purpose

Removes a semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>

int msem_remove ( Sem)
msemaphore *Sem;
```

Description

The **msem_remove** subroutine removes a binary semaphore. Any subsequent use of the **msemaphore** structure before it is again initialized by calling the **msem_init** subroutine will have undefined results.

The **msem_remove** subroutine also causes any process waiting in the **msem_lock** subroutine on the removed semaphore to return with an error.

If the **msemaphore** structure contains any value not resulting from a call to the **msem_init** subroutine, followed by a (possibly empty) sequence of calls to the **msem_lock** and **msem_unlock** subroutines, the result is undefined. The address of an **msemaphore** structure is significant. If the **msemaphore** structure contains any value copied from an **msemaphore** structure at a different address, the result is undefined.

Parameters

Item	Description
Sem	Points to an **msemaphore** structure that specifies the semaphore to be removed.

Return Values

When successful, the **msem_remove** subroutine returns a value of 0. Otherwise, it returns a -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **msem_remove** subroutine is unsuccessful, the **errno** global variable is set to the following value:

Item	Description
EINVAL	Indicates the *Sem* parameter points to an **msemaphore** structure that specifies a semaphore that has been removed.

msem_unlock Subroutine

Purpose

Unlocks a semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>

int msem_unlock ( Sem,  Condition)
msemaphore *Sem;
int Condition;
```

Description

The **msem_unlock** subroutine attempts to unlock a binary semaphore.

If the semaphore is currently locked, it is unlocked and the **msem_unlock** subroutine completes successfully.

If the *Condition* parameter is 0, the semaphore is unlocked, regardless of whether or not any other processes are currently attempting to lock it. If the *Condition* parameter is set to the **MSEM_IF_WAITERS** value, and another process is waiting to lock the semaphore or it cannot be reliably determined whether some process is waiting to lock the semaphore, the semaphore is unlocked by the calling process. If the *Condition* parameter is set to the **MSEM_IF_WAITERS** value and no process is waiting to lock the semaphore, the semaphore will not be unlocked and an error will be returned.

Parameters

Item	Description
Sem	Points to an **msemaphore** structure that specifies the semaphore to be unlocked.
Condition	Determines whether the **msem_unlock** subroutine unlocks the semaphore if no other processes are waiting to lock it.

Return Values

When successful, the **msem_unlock** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **msem_unlock** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

Item	Description
EAGAIN	Indicates a *Condition* value of **MSEM_IF_WAITERS** was specified and there were no waiters.
EINVAL	Indicates the *Sem* parameter points to an **msemaphore** structure specifying a semaphore that has been removed, or the *Condition* parameter is not valid.

msgctl Subroutine

Purpose

Provides message control operations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/msg.h>

int msgctl (MessageQueueID,Command,Buffer)
int  MessageQueueID,  Command;
struct msqid_ds * Buffer;
```

Description

The **msgctl** subroutine provides a variety of message control operations as specified by the *Command* parameter and stored in the structure pointed to by the *Buffer* parameter. The **msqid_ds** structure is defined in the **sys/msg.h** file.

The following limits apply to the message queue:

- Maximum message size is 4 Megabytes.
- Maximum number of messages per queue is 524288.
- Maximum number of message queue IDs is 1048576.
- Maximum number of bytes in a queue is 4 Megabytes.

Parameters

Item	Description
MessageQueueID	Specifies the message queue identifier.

Command	The following values for the *Command* parameter are available:
	IPC_STAT
	Stores the current value of the above fields of the data structure associated with the *MessageQueueID* parameter into the **msqid_ds** structure pointed to by the *Buffer* parameter.
	The current process must have read permission in order to perform this operation.
	IPC_SET
	Sets the value of the following fields of the data structure associated with the *MessageQueueID* parameter to the corresponding values found in the structure pointed to by the *Buffer* parameter:
	`msg_perm.uid`
	`msg_perm.gid`
	`msg_perm.mode/*Only the low-order`
	`nine bits*/`
	`msg_qbytes`
	The effective user ID of the current process must have root user authority or must equal the value of the `msg_perm.uid` or `msg_perm.cuid` field in the data structure associated with the *MessageQueueID* parameter in order to perform this operation. To raise the value of the `msg_qbytes` field, the effective user ID of the current process must have root user authority.
	IPC_RMID
	Removes the message queue identifier specified by the *MessageQueueID* parameter from the system and destroys the message queue and data structure associated with it. The effective user ID of the current process must have root user authority or be equal to the value of the `msg_perm.uid` or `msg_perm.cuid` field in the data structure associated with the *MessageQueueID* parameter to perform this operation.

Buffer	Points to a **msqid_ds** structure.

Return Values

Upon successful completion, the **msgctl** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgctl** subroutine is unsuccessful if any of the following conditions is true:

Item	Description
EINVAL	The *Command* or *MessageQueueID* parameter is not valid.

Item	Description
EACCES	The *Command* parameter is equal to the **IPC_STAT** value, and the calling process was denied read permission.
EPERM	The *Command* parameter is equal to the **IPC_RMID** value and the effective user ID of the calling process does not have root user authority. Or, the *Command* parameter is equal to the **IPC_SET** value, and the effective user ID of the calling process is not equal to the value of the `msg_perm.uid` field or the `msg_perm.cuid` field in the data structure associated with the *MessageQueueID* parameter.
EPERM	The *Command* parameter is equal to the **IPC_SET** value, an attempt was made to increase the value of the `msg_qbytes` field, and the effective user ID of the calling process does not have root user authority.
EFAULT	The *Buffer* parameter points outside of the process address space.

msgget Subroutine

Purpose

Gets a message queue identifier.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/msg.h>

int msgget ( Key,  MessageFlag)
key_t Key;
int MessageFlag;
```

Description

The **msgget** subroutine returns the message queue identifier associated with the specified *Key* parameter.

A message queue identifier, associated message queue, and data structure are created for the value of the *Key* parameter if one of the following conditions is true:

- The *Key* parameter is equal to the **IPC_PRIVATE** value.
- The *Key* parameter does not already have a message queue identifier associated with it, and the **IPC_CREAT** value is set.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- The `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` fields are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of the `msg_perm.mode` field are set equal to the low-order 9 bits of the *MessageFlag* parameter.
- The `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` fields are set equal to 0.
- The `msg_ctime` field is set equal to the current time.
- The `msg_qbytes` field is set equal to the system limit.

The **msgget** subroutine performs the following actions:

- The **msgget** subroutine either finds or creates (depending on the value of the *MessageFlag* parameter) a queue with the *Key* parameter.
- The **msgget** subroutine returns the ID of the queue header to its caller.

Limits on message size and number of messages in the queue can be found in *General Programming Concepts: Writing and Debugging Programs*.

Parameters

Item	Description
Key	Specifies either the value **IPC_PRIVATE** or an Interprocess Communication (IPC) key constructed by the **ftok** subroutine (or by a similar algorithm).
MessageFlag	Constructed by logically ORing one or more of the following values: **IPC_CREAT** Creates the data structure if it does not already exist. **IPC_EXCL** Causes the **msgget** subroutine to fail if the **IPC_CREAT** value is also set and the data structure already exists. **S_IRUSR** Permits the process that owns the data structure to read it. **S_IWUSR** Permits the process that owns the data structure to modify it. **S_IRGRP** Permits the group associated with the data structure to read it. **S_IWGRP** Permits the group associated with the data structure to modify it. **S_IROTH** Permits others to read the data structure. **S_IWOTH** Permits others to modify the data structure. Values that begin with **S_I** are defined in the **sys/mode.h** file and are a subset of the access permissions that apply to files.

Return Values

Upon successful completion, the **msgget** subroutine returns a message queue identifier. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgget** subroutine is unsuccessful if any of the following conditions is true:

Item	Description
EACCES	A message queue identifier exists for the *Key* parameter, but operation permission as specified by the low-order 9 bits of the *MessageFlag* parameter is not granted.
ENOENT	A message queue identifier does not exist for the *Key* parameter and the **IPC_CREAT** value is not set.
ENOSPC	A message queue identifier is to be created, but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

Item	Description
EEXIST	A message queue identifier exists for the *Key* parameter, and both **IPC_CREAT** and **IPC_EXCL** values are set.

msgrcv Subroutine

Purpose

Reads a message from a queue.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/msg.h>
```

```
int msgrcv (MessageQueueID, MessagePointer,MessageSize,MessageType, MessageFlag)
int  MessageQueueID,  MessageFlag;
void * MessagePointer;
size_t  MessageSize;
long int  MessageType;
```

Description

The **msgrcv** subroutine reads a message from the queue specified by the *MessageQueueID* parameter and stores it into the structure pointed to by the *MessagePointer* parameter. The current process must have read permission in order to perform this operation.

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

Limits on message size and number of messages in the queue can be found in *General Programming Concepts: Writing and Debugging Programs*.

Note: For a 64-bit process, the **mtype** field is 64 bits long. However, for compatibility with 32-bit processes, the **mtype** field must be a 32-bit signed value that is sign-extended to 64 bits. The most significant 32 bits are not put on the message queue. For a 64-bit process, the **mtype** field is again sign-extended to 64 bits.

Parameters

Item	Description
MessageQueueID	Specifies the message queue identifier.

MessagePointer	Points to a **msgbuf** structure containing the message. The **msgbuf** structure is defined in the **sys/msg.h** file and contains the following fields:
	`mtyp_t  mtype;  /* Message type */`
	`char  mtext[1];  /* Beginning of message text */`
	The `mtype` field contains the type of the received message as specified by the sending process. The `mtext` field is the text of the message.

Item	Description
MessageSize	Specifies the size of the `mtext` field in bytes. The received message is truncated to the size specified by the *MessageSize* parameter if it is longer than the size specified by the *MessageSize* parameter and if the **MSG_NOERROR** value is set in the *MessageFlag* parameter. The truncated part of the message is lost and no indication of the truncation is given to the calling process.
MessageType	Specifies the type of message requested as follows: • If equal to the value of 0, the first message on the queue is received. • If greater than 0, the first message of the type specified by the *MessageType* parameter is received. • If less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *MessageType* parameter is received.
MessageFlag	Specifies either a value of 0 or is constructed by logically ORing one or more of the following values: **MSG_NOERROR** Truncates the message if it is longer than the *MessageSize* parameter. **IPC_NOWAIT** Specifies the action to take if a message of the desired type is not on the queue: • If the **IPC_NOWAIT** value is set, the calling process returns a value of -1 and sets the **errno** global variable to the **ENOMSG** error code. • If the **IPC_NOWAIT** value is not set, the calling process suspends execution until one of the following occurs: – A message of the desired type is placed on the queue. – The message queue identifier specified by the *MessageQueueID* parameter is removed from the system. When this occurs, the **errno** global variable is set to the **EIDRM** error code, and a value of -1 is returned. – The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner described in the **sigaction** subroutine.

Return Values

Upon successful completion, the **msgrcv** subroutine returns a value equal to the number of bytes actually stored into the `mtext` field and the following actions are taken with respect to fields of the data structure associated with the *MessageQueueID* parameter:

- The `msg_qnum` field is decremented by 1.
- The `msg_lrpid` field is set equal to the process ID of the calling process.
- The `msg_rtime` field is set equal to the current time.

If the **msgrcv** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgrcv** subroutine is unsuccessful if any of the following conditions is true:

Item	Description
EINVAL	The *MessageQueueID* parameter is not a valid message queue identifier.
EACCES	The calling process is denied permission for the specified operation.

Item	Description
E2BIG	The `mtext` field is greater than the *MessageSize* parameter, and the **MSG_NOERROR** value is not set.
ENOMSG	The queue does not contain a message of the desired type and the **IPC_NOWAIT** value is set.
EFAULT	The *MessagePointer* parameter points outside of the allocated address space of the process.
EINTR	The **msgrcv** subroutine is interrupted by a signal.
EIDRM	The message queue identifier specified by the *MessageQueueID* parameter has been removed from the system.

msgsnd Subroutine

Purpose

Sends a message.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/msg.h>
```

```
int msgsnd (MessageQueueID, MessagePointer,MessageSize, MessageFlag)
int  MessageQueueID,  MessageFlag;
const void * MessagePointer;
size_t  MessageSize;
```

Description

The **msgsnd** subroutine sends a message to the queue specified by the *MessageQueueID* parameter. The current process must have write permission to perform this operation. The *MessagePointer* parameter points to an **msgbuf** structure containing the message. The **sys/msg.h** file defines the **msgbuf** structure. The structure contains the following fields:

```
mtyp_t   mtype;      /* Message type */
char     mtext[1];   /* Beginning of message text */
```

The `mtype` field specifies a positive integer used by the receiving process for message selection. The `mtext` field can be any text of the length in bytes specified by the *MessageSize* parameter. The *MessageSize* parameter can range from 0 to the maximum limit imposed by the system.

The following example shows a typical user-defined **msgbuf** structure that includes sufficient space for the largest message:

```
struct my_msgbuf
mtyp_t     mtype;
char       mtext[MSGSIZ]; /* MSGSIZ is the size of the largest message */
```

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

The following system limits apply to the message queue:

- Maximum message size is 4 Megabytes.

- Maximum number of messages per queue is 524288.
- Maximum number of message queue IDs is 131072
- Maximum number of bytes in a queue is 4 Megabytes.

Note: For a 64-bit process, the **mtype** field is 64 bits long. However, for compatibility with 32-bit processes, the **mtype** field must be a 32-bit signed value that is sign-extended to 64 bits. The most significant 32 bits are not put on the message queue. For a 64-bit process, the **mtype** field is again sign-extended to 64 bits.

The *MessageFlag* parameter specifies the action to be taken if the message cannot be sent for one of the following reasons:

- The number of bytes already on the queue is equal to the number of bytes defined by the**msg_qbytes** structure.
- The total number of messages on the queue is equal to a system-imposed limit.

These actions are as follows:

- If the *MessageFlag* parameter is set to the **IPC_NOWAIT** value, the message is not sent, and the **msgsnd** subroutine returns a value of -1 and sets the **errno** global variable to the **EAGAIN** error code.
- If the *MessageFlag* parameter is set to 0, the calling process suspends execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The *MessageQueueID* parameter is removed from the system. (For information on how to remove the *MessageQueueID* parameter, see the **msgctl**. When this occurs, the **errno** global variable is set equal to the **EIDRM** error code, and a value of -1 is returned.
  - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in the **sigaction** subroutine.

Parameters

Item	Description
MessageQueueID	Specifies the queue to which the message is sent.
MessagePointer	Points to a **msgbuf** structure containing the message.
MessageSize	Specifies the length, in bytes, of the message text.
MessageFlag	Specifies the action to be taken if the message cannot be sent.

Return Values

Upon successful completion, a value of 0 is returned and the following actions are taken with respect to the data structure associated with the *MessageQueueID* parameter:

- The `msg_qnum` field is incremented by 1.
- The `msg_lspid` field is set equal to the process ID of the calling process.
- The `msg_stime` field is set equal to the current time.

If the **msgsnd** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgsnd** subroutine is unsuccessful and no message is sent if one or more of the following conditions is true:

Item	Description
EACCES	The calling process is denied permission for the specified operation.
EAGAIN	The message cannot be sent for one of the reasons stated previously, and the *MessageFlag* parameter is set to the **IPC_NOWAIT** value or the system has temporarily ran out of memory resource.
EFAULT	The *MessagePointer* parameter points outside of the address space of the process.
EIDRM	The message queue identifier specified by the *MessageQueueID* parameter has been removed from the system.
EINTR	The **msgsnd** subroutine received a signal.
EINVAL	The *MessageQueueID* parameter is not a valid message queue identifier.
EINVAL	The `mtype` field is less than 1.
EINVAL	The *MessageSize* parameter is less than 0 or greater than the system-imposed limit.
EINVAL	The upper 32-bits of the 64-bit mtype field for a 64-bit process is not 0.
ENOMEM	The message could not be sent because not enough storage space was available.

msgxrcv Subroutine

Purpose

Receives an extended message.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/msg.h>

int msgxrcv (MessageQueueID, MessagePointer, MessageSize, MessageType,
MessageFlag) int MessageQueueID, MessageFlag; size_t MessageSize; struct
msgxbuf * MessagePointer; long MessageType;
```

Description

The **msgxrcv** subroutine reads a message from the queue specified by the *MessageQueueID* parameter and stores it into the extended message receive buffer pointed to by the *MessagePointer* parameter. The current process must have read permission in order to perform this operation. The **msgxbuf** structure is defined in the **sys/msg.h** file.

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

The following limits apply to the message queue:

- Maximum message size is 4 Megabytes.
- Maximum number of messages per queue is 8192.
- Maximum number of message queue IDs is 131072.
- Maximum number of bytes in a queue is 4 Megabytes.

Note: For a 64-bit process, the **mtype** field is 64 bits long. However, for compatibility with 32-bit processes, the **mtype** field must be a 32-bit signed value that is sign-extended to 64 bits. The most

significant 32 bits are not put on the message queue. For a 64-bit process, the **mtype** field is again sign-extended to 64 bits.

Parameters

Item	Description
MessageQueueID	Specifies the message queue identifier.
MessagePointer	Specifies a pointer to an extended message receive buffer where a message is stored.
MessageSize	Specifies the size of the `mtext` field in bytes. The receive message is truncated to the size specified by the *MessageSize* parameter if it is larger than the *MessageSize* parameter and the **MSG_NOERROR** value is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than the number of bytes specified by the *MessageSize* parameter and the **MSG_NOERROR** value is not set, the **msgxrcv** subroutine is unsuccessful and sets the **errno** global variable to the **E2BIG** error code.
MessageType	Specifies the type of message requested as follows: • If the *MessageType* parameter is equal to 0, the first message on the queue is received. • If the *MessageType* parameter is greater than 0, the first message of the type specified by the *MessageType* parameter is received. • If the *MessageType* parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *MessageType* parameter is received.
MessageFlag	Specifies a value of 0 or a value constructed by logically ORing one or more of the following values: **MSG_NOERROR** Truncates the message if it is longer than the number of bytes specified by the *MessageSize* parameter. **IPC_NOWAIT** Specifies the action to take if a message of the desired type is not on the queue: • If the **IPC_NOWAIT** value is set, the calling process returns a value of -1 and sets the **errno** global variable to the **ENOMSG** error code. • If the **IPC_NOWAIT** value is not set, the calling process suspends execution until one of the following occurs: – A message of the desired type is placed on the queue. – The message queue identifier specified by the *MessageQueueID* parameter is removed from the system. When this occurs, the **errno** global variable is set to the **EIDRM** error code, and a value of -1 is returned. – The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner prescribed in the **sigaction** subroutine.

Return Values

Upon successful completion, the **msgxrcv** subroutine returns a value equal to the number of bytes actually stored into the `mtext` field, and the following actions are taken with respect to the data structure associated with the *MessageQueueID* parameter:

- The `msg_qnum` field is decremented by 1.
- The `msg_lrpid` field is set equal to the process ID of the calling process.
- The `msg_rtime` field is set equal to the current time.

If the **msgxrcv** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **msgxrcv** subroutine is unsuccessful if any of the following conditions is true:

Item	Description
EINVAL	The *MessageQueueID* parameter is not a valid message queue identifier.
EACCES	The calling process is denied permission for the specified operation.
EINVAL	The *MessageSize* parameter is less than 0.
E2BIG	The `mtext` field is greater than the *MessageSize* parameter, and the **MSG_NOERROR** value is not set.
ENOMSG	The queue does not contain a message of the desired type and the **IPC_NOWAIT** value is set.
EFAULT	The *MessagePointer* parameter points outside of the process address space.
EINTR	The **msgxrcv** subroutine was interrupted by a signal.
EIDRM	The message queue identifier specified by the *MessageQueueID* parameter is removed from the system.

msleep Subroutine

Purpose

Puts a process to sleep when a semaphore is busy.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>

int msleep (Sem)
msemaphore * Sem;
```

Description

The **msleep** subroutine puts a calling process to sleep when a semaphore is busy. The semaphore should be located in a shared memory region. Use the **mmap** subroutine to create the shared memory section.

All of the values in the **msemaphore** structure must result from a **msem_init** subroutine call. This call may or may not be followed by a sequence of calls to the **msem_lock** subroutine or the **msem_unlock**

subroutine. If the **msemaphore** structure value originates in another manner, the results of the **msleep** subroutine are undefined.

The address of the **msemaphore** structure is significant. You should be careful not to modify the structure's address. If the structure contains values copied from a **msemaphore** structure at another address, the results of the **msleep** subroutine are undefined.

Parameters

Item	Description
Sem	Points to the **msemaphore** structure that specifies the semaphore.

Error Codes

If the **msleep** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

Item	Description
EFAULT	Indicates that the *Sem* parameter points to an invalid address or the address does not contain a valid **msemaphore** structure.
EINTR	Indicates that the process calling the **msleep** subroutine was interrupted by a signal while sleeping.

msync Subroutine

Purpose

Synchronize memory with physical storage.

Library

Standard C Library (**libc.a**).

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>

int msync ( addr,  len,  flags)
void *addr;
size_t len;
int flags;
```

Description

The **msync** subroutine controls the caching operations of a mapped file or shared memory region. Use the **msync** subroutine to transfer modified pages in the region to the underlying file storage device.

If the application has requested Single UNIX Specification, Version 2 compliant behavior, then the mapped file's last data modification and last file status change timestamps are marked for update upon successful completion of the **msync** subroutine call if the file has been modified.

Parameters

Item	Description
addr	Specifies the address of the region to be synchronized. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
len	Specifies the length, in bytes, of the region to be synchronized. If the *len* parameter is not a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region is rounded up to the next multiple of the page size.
flags	Specifies one or more of the following symbolic constants that determine the way caching operations are performed: **MS_SYNC** Specifies synchronous cache flush. The **msync** subroutine does not return until the system completes all I/O operations. This flag is invalid when the **MAP_PRIVATE** flag is used with the **mmap** subroutine. **MAP_PRIVATE** is the default privacy setting. When the **MS_SYNC** and **MAP_PRIVATE** flags both are used, the **msync** subroutine returns an **errno** value of **EINVAL**. **MS_ASYNC** Specifies an asynchronous cache flush. The **msync** subroutine returns after the system schedules all I/O operations. This flag is invalid when the **MAP_PRIVATE** flag is used with the **mmap** subroutine. **MAP_PRIVATE** is the default privacy setting. When the **MS_SYNC** and **MAP_PRIVATE** flags both are used, the **msync** subroutine returns an **errno** value of **EINVAL**. **MS_INVALIDATE** Specifies that the **msync** subroutine invalidates all cached copies of the pages. New copies of the pages must then be obtained from the file system the next time they are referenced.

Return Values

When successful, the **msync** subroutine returns 0. Otherwise, it returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **msync** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

Item	Description
EBUSY	One or more pages in the range passed to the **msync** subroutine is pinned.
EIO	An I/O error occurred while reading from or writing to the file system.
ENOMEM	The range specified by (*addr*, *addr* + *len*) is invalid for a process' address space, or the range specifies one or more unmapped pages.
EINVAL	The *addr* argument is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, or the *flags* parameter is invalid. The address of the region is within the process' inheritable address space.

mt__trce Subroutine

Purpose

Dumps traceback information into a lightweight core file.

Library

PTools Library (**libptools_ptr.a**)

Syntax

```
void mt__trce (int  FileDescriptor, int Signal, struct sigcontext *Context, int
Node);
```

Description

The **mt__trce** subroutine dumps traceback information of the calling thread and all other threads allocated in the process space into the file specified by the *FileDescriptor* parameter. The format of the output from this subroutine complies with the Parallel Tools Consortium Lightweight CoreFile Format. Threads, except the calling thread, will be suspended after the calling thread enters this subroutine and while the traceback information is being obtained. Threads execution resumes when this subroutine returns.

When using the **mt__trce** subroutine in a signal handler, it is recommended that the application be started with the environment variable AIXTHREAD_SCOPE set to S (As in export AIXTHREAD_SCOPE=S). If this variable is not set, the application may hang.

Parameters

Item	Description
Context	Points to the **sigcontext** structure containing the context of the thread when the signal happens. The context is used to generate the traceback information for the calling thread. This is used only if the *Signal* parameter is nonzero. If the **mt__trce** subroutine is called with the *Signal* parameter set to zero, the *Context* parameter is ignored and the traceback information is generated based on the current context of the calling thread. Refer to the **sigaction** subroutine for further description about signal handlers and how the **sigcontext** structure is passed to a signal handler.
File Descriptor	The file descriptor of the lightweight core file. It specifies the target file into which the traceback information is written.
Node	Specifies the number of the tasks or nodes where this subroutine is executing and is used only for a parallel application consisting of multiple tasks. The *Node* parameter will be used in section headers of the traceback information to identify the task or node from which the information is generated.
Signal	The number of the signal that causes the signal handler to be executed. This is used only if the **mt__trce** subroutine is called from a signal handler. A Fault-Info section defined by the Parallel Tools Consortium Lightweight Core File Format will be written into the output lightweight core file based on this signal number. If the **mt__trce** subroutine is not called from a signal handler, the *Signal* parameter must be set to 0 and a Fault-Info section will not be generated.

Note:

1. To obtain source line information in the traceback, the programs must have been compiled with the **-g** option to include the necessary line number information in the executable files. Otherwise, address offset from the beginning of the function is provided.
2. Line number information is not provided for shared objects even if they were compiled with the **-g** option.
3. Function names are not provided if a program or a library is compiled with optimization. To obtain function name information in the traceback and still have the object code optimized, compiler option **-qtbtable=full** must be specified.

4. In rare cases, the traceback of a thread may seem to skip one level of procedure calls. This is because the traceback is obtained at the moment the thread entered a procedure and has not yet allocated a stack frame.
5. The source line information in a `Lightweight_core` file is not displayed by default when the text page size is 64 K. When the text page size is 64K, use the environment variable `AIX_LDSYM=ON` to get the source line information in a `Lightweight_core` file.

Return Values

Upon successful completion, the **mt__trce** subroutine returns a value of 0. Otherwise, an error number is returned to indicate the error.

Error Codes

If an error occurs, the subroutine returns -1 and the **errno** global variable is set to indicate the error, as follows:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor open for writing.
ENOSPC	No free space is left in the file system containing the file.
EDQUOT	New disk blocks cannot be allocated for the file because the user or group quota of blocks has been exhausted on the file system.
EINVAL	The value of the *Signal* parameter is invalid or the *Context* parameter points to an invalid context.
ENOMEM	Insufficient memory exists to perform the operation.

Examples

1. The following example calls the **mt__trce** subroutine to generate traceback information in a signal handler.

```
void
my_handler(int signal,
           int code,
           struct sigcontext *sigcontext_data)
{
    int lcf_fd;
    ....
    lcf_fd = open(file_name, O_WRONLY|O_CREAT|O_APPEND, 0666);
    ....
    rc = mt__trce(lcf_fd, signal, sigcontext_data, 0);
    ....
    close(lcf_fd);
    ....
}
```

2. The following is an example of the lightweight core file generated by the **mt__trce** subroutine. Notice the thread ID in the information is the unique sequence number of a thread for the life time of the process containing the thread.

```
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.0
+++LCB 1.0 Thu Jun 30 16:02:35 1999 Generated by AIX
#
+++ID Node 0 Process 21084 Thread 1
***FAULT "SIGABRT - Abort"
+++STACK
func2 : 123 # in file
func1 : 272 # in file
main : 49 # in file
---STACK
---ID Node 0 Process 21084 Thread 1
#
+++ID Node 0 Process 21084 Thread 2
```

```
+++STACK
nsleep : 0x0000001c
sleep : 0x00000030
f_mt_exec : 21 # in file
_pthread_body : 0x00000114
---STACK
---ID Node 0 Process 21084 Thread 2
#
+++ID Node 0 Process 21084 Thread 3
+++STACK
nsleep : 0x0000001c
sleep : 0x00000030
f_mt_exec : 21 # in file
_pthread_body : 0x00000114
---STACK
---ID Node 0 Process 21084 Thread 3
---LCB
```

mtx_destroy, mtx_init, mtx_lock, mtx_timedlock, mtx_trylock, and mtx_unlock Subroutine

Purpose

The **mtx_destroy** subroutine releases any resources that are used by the **mtx** mutex variable.

The **mtx_init** subroutine creates a **mtx** mutex variable that has the properties specified by the **type** parameter.

The **mtx_lock** and **mtx_unlock** subroutine locks and unlocks the **mtx** mutex variable.

The **mtx_timedlock** subroutine locks the **mtx** mutex variable for the time that is specified by the **tsun** parameter.

The **mtx_trylock** subroutine tries to lock the **mtx** mutex variable, if available.

Library

Standard C library (**libc.a**)

Syntax

```
#include <threads.h>
void mtx_destroy (mtx_t * mtx);

int mtx_init (mtx_t * mtx, int type);

int mtx_lock (mtx_t * mtx);

int mtx_init (mtx_t * mtx, int type);

int mtx_timedlock (mtx_t * restrict mtx, const struct timespec * restrict  ts);

int mtx_trylock (mtx_t * mtx);
```

Description

The **mtx_destroy** subroutine releases any resources that are used by the mutex variable specified by the **mtx** parameter.

The **mtx_destroy** subroutine requires that threads are not blocked while waiting for the mutex variable specified by the **mtx** parameter.

The **mtx_init** subroutine creates a mutex object that has the **type** parameter, which can accept any one of the following values:

- **mtx_plain** for a simple nonrecursive mutex
- **mtx_timed** for a nonrecursive mutex that supports timeout

- **mtx_plain** or **mtx_recursive** for a simple recursive mutex
- **mtx_timed** or **mtx_recursive** for a recursive mutex that supports timeout

If the **mtx_init** subroutine is successful, it sets the mutex variable specified by the **mtx** parameter to a value that uniquely identifies the newly created mutex.

The **mtx_lock** subroutine locks the mutex variable specified by the **mtx** parameter. If the mutex variable is nonrecursive, it is not locked by the calling thread.

The **mtx_timedlock** subroutine tries to lock the mutex variable specified by the **mtx** parameter or till the **TIME_UTC** based calendar time is pointed to by the value that is specified in the **ts** parameter. The specified mutex variable supports timeout operation.

The **mtx_trylock** subroutine tries to lock the mutex variable specified by the **mtx** parameter. If the mutex is already locked, the function returns without blocking the mutex variable.

Previous calls to the **mtx_unlock** subroutine on the same mutex synchronizes the operations while using any of the subroutines, such as the **mtx_lock**, **mtx_trylock** or **mtx_timedlock** subroutines.

The **mtx_unlock** subroutine unlocks the mutex variable specified by the **mtx** parameter. The mutex specified by the **mtx** parameter is locked by the calling thread.

Parameters

Item	Description
mtx	Specifies the mutex variable to be created and locked. It also specifies the mutex variable for which the resources are to be released based on the type of the subroutine in which the parameter is referenced.
type	Specifies the properties of the mutex variable and contains the combination of any of the following values: **mtx_plain**, **mtx_timed**, or **mtx_recursive**.
ts	Specifies the maximum time for the **mtx_timedlock** subroutine to block the mutex variable.

Return Values

The **mtx_destroy** subroutine returns no value.

The **mtx_init**, **mtx_lock** and **mtx_unlock** subroutines return the value of **thrd_success** on success, and returns the value of **thrd_error** if the request cannot be processed.

The **mtx_timedlock** subroutine returns the value of **thrd_success** on success.

The **mtx_timedlock** subroutine returns the value of **thrd_timedout** if the specified time is reached without acquiring the requested resource.

The **mtx_timedlock** subroutine returns the value of **thrd_error** if the request cannot be processed.

The **mtx_trylock** subroutine returns the value of **thrd_success** on success, it returns the value of **thrd_busy** if the requested resource is already in use, and it returns the value of **thrd_error** if the request cannot be processed.

Files

The **threads.h** file defines standard macros, data types, and subroutines.

munmap Subroutine

Purpose

Unmaps pages of memory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/mman.h>
```

```
int munmap ( addr,  len)
void *addr;
size_t len;
```

Description

The **munmap** subroutine unmaps a mapped file or shared memory region or anonymous memory region. The **munmap** subroutine unmaps regions created from calls to the **mmap** subroutine only.

If an address lies in a region that is unmapped by the **munmap** subroutine and that region is not subsequently mapped again, any reference to that address will result in the delivery of a **SIGSEGV** signal to the process.

Parameters

Item	Description
addr	Specifies the address of the region to be unmapped. Must be a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
len	Specifies the length, in bytes, of the region to be unmapped. If the *len* parameter is not a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter, the length of the region is rounded up to the next multiple of the page size.

Return Values

When successful, the **munmap** subroutine returns 0. Otherwise, it returns -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **munmap** subroutine is unsuccessful, the **errno** global variable is set to the following value:

Item	Description
EINVAL	The *addr* parameter is not a multiple of the page size as returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
EINVAL	The application has requested Single UNIX Specification, Version 2 compliant behavior and the *len* arguement is 0.

mvcur Subroutine

Purpose

Output cursor movement commands to the terminal.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int mvcur(int oldrow,
int oldcol,
int newrow,
int newcol);
```

Description

The **mvcur** subroutine outputs one or more commands to the terminal that move the terminal's cursor to (*newrow*, *newcol*), an absolute position on the terminal screen. The (*oldrow*, *oldcol*) arguments specify the former cursor position. Specifying the former position is necessary on terminals that do not provide coordinate-based movement commands. On terminals that provide these commands, Curses may select a more efficient way to move the cursor based on the former position. If (*newrow*, *newcol*) is not a valid address for the terminal in use, the **mvcur** subroutine fails. If (*oldrow*, *oldcol*) is the same as (*newrow*, *newcol*), **mvcur** succeeds without taking any action. If **mvcur** outputs a cursor movement command, it updates its information concerning the location of the cursor on the terminal.

Parameters

Item	Description
newcol	Holds the new column coordinate of the physical cursor.
newrow	Holds the new row coordinate of the physical cursor.
oldcol	Holds the old column coordinate of the physical cursor.
oldrow	Holds the old row coordinate of the physical cursor.

Return Values

Upon successful completion, the **mvcur** subroutine returns OK. Otherwise, it returns ERR.

Examples

1. To move the physical cursor from the coordinates y = 5, x = 15 to y = 25, x = 30, use:

   ```
   mvcur(5, 15, 25, 30);
   ```

2. To move the physical cursor from unknown coordinates to y = 5, x = 0, use:

   ```
   mvcur(50, 50, 5, 0);
   ```

   In this example, the physical cursor's current coordinates are unknown. Therefore, arbitrary values are assigned to the *OldLine* and *OldColumn* parameters and the desired coordinates are assigned to the *NewLine* and *NewColumn* parameters. This is called an *absolute move*.

mvwin Subroutine

Purpose

Moves a window or subwindow to the specified coordinates.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int mvwin
(WINDOW *win,
int y,
int x);
```

Description

The **mvwin** subroutine moves the specified window so that its origin is at position (*y*, *x*). If the move causes any portion of the window to extend past any edge of the screen, the function fails and the window is not moved.

Parameters

Item	Description
**win*	
x	
y	

Return Values

Upon successful completion, the **mvwin** subroutine returns OK. Otherwise, it returns ERR.

Examples

1. To move the user-defined window `my_window` from its present location to the upper left corner of the terminal, enter:

   ```
   WINDOW *my_window;
   mvwin(my_window, 0, 0);
   ```

2. To move the user-defined window `my_window` from its present location to the coordinates y = `20`, x = `10`, enter:

   ```
   WINDOW *my_window;
   mvwin(my_window, 20, 10);
   ```

mwakeup Subroutine

Purpose

Wakes up a process that is waiting on a semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>
int mwakeup (Sem)
msemaphore * Sem;
```

Description

The **mwakeup** subroutine wakes up a process that is sleeping and waiting for an idle semaphore. The semaphore should be located in a shared memory region. Use the **mmap** subroutine to create the shared memory section.

All of the values in the **msemaphore** structure must result from a **msem_init** subroutine call. This call may or may not be followed by a sequence of calls to the **msem_lock** subroutine or the **msem_unlock** subroutine. If the **msemaphore** structure value originates in another manner, the results of the **mwakeup** subroutine are undefined.

The address of the **msemaphore** structure is significant. You should be careful not to modify the structure's address. If the structure contains values copied from a **msemaphore** structure at another address, the results of the **mwakeup** subroutine are undefined.

Parameters

Item	Description
Sem	Points to the **msemaphore** structure that specifies the semaphore.

Return Values

When successful, the **mwakeup** subroutine returns a value of 0. Otherwise, this routine returns a value of -1 and sets the **errno** global variable to **EFAULT**.

Error Codes

A value of **EFAULT** indicates that the *Sem* parameter points to an invalid address or that the address does not contain a valid **msemaphore** structure.

n

The following Base Operating System (BOS) runtime services begin with the letter *n*.

nan, nanf, nanl, nand32, nand64, and nand128 Subroutines

Purpose

Return a quiet NaN.

Syntax

```
#include <math.h>

double nan (tagp)
const char *tagp;

float nanf (tagp)
const char *tagp;

long double nanl (tagp)
const char *tagp;

_Decimal32 nand32(tagp)
const char *tagp;

_Decimal64 nand64(tagp)
const char *tagp;

_Decimal128 nand128(tagp)
const char *tagp;
```

Description

The function call **nan***("n-char-sequence")* is equivalent to:

```
strtod("NAN(n-char-sequence)", (char **) NULL);
```

The function call **nan**(" ") is equivalent to:

```
strtod("NAN()", (char **) NULL)
```

If *tagp* does not point to an *n*-**char** sequence or an empty string, the function call is equivalent to:

```
strtod("NAN", (char **) NULL)
```

Function calls to the **nanf**, **nanl**, **nand32**, **nand64**, and **nand128** subroutines are equivalent to the corresponding function calls to the **strtof**, **strtold**, **strtod32**, **strtod64**, and **strtod128** subroutines.

Parameters

Item	Description
tagp	Indicates the content of the quiet NaN.

Return Values

The **nan**, **nanf**, **nanl**, **nand32**, **nand64**, and **nand128** subroutines return a quiet NaN with content indicated through *tagp*.

nanosleep Subroutine

Purpose

Causes the current thread to be suspended from execution.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>

int nanosleep (rqtp, rmtp)
const struct timespec *rqtp;
struct timespec *rmtp;
```

Description

The **nanosleep** subroutine causes the current thread to be suspended from execution until either the time interval specified by the *rqtp* parameter has elapsed or a signal is delivered to the calling thread and its action is to invoke a signal-catching function or to terminate the process. The suspension time may be longer than requested because the argument value is rounded up to an integer multiple of the sleep resolution. This can also occur because of the scheduling of other activity by the system. Unless it is interrupted by a signal, the suspension time will not be less than the time specified by the *rqtp* parameter, as measured by the system clock **CLOCK_REALTIME**.

The use of the **nanosleep** subroutine has no effect on the action or blockage of any signal.

Parameters

Item	Description
rqtp	Specifies the time interval that the thread is suspended.
rmtp	Points to the **timespec** structure.

Return Values

If the **nanosleep** subroutine returns because the requested time has elapsed, its return value is zero.

If the **nanosleep** subroutine returns because it has been interrupted by a signal, it returns -1 and sets **errno** to indicate the interruption. If the *rmtp* parameter is non-NULL, the **timespec** structure is updated to contain the amount of time remaining in the interval (the requested time minus the time actually slept). If the *rmtp* parameter is NULL, the remaining time is not returned.

If the **nanosleep** subroutine fails, it returns -1 and sets **errno** to indicate the error.

Error Codes

The **nanosleep** subroutine fails if:

Item	Description
EINTR	The **nanosleep** subroutine was interrupted by a signal.
EINVAL	The *rqtp* parameter specified a nanosecond value less than zero or greater than or equal to 1000 million.

nearbyint, nearbyintf, nearbyintl, nearbyintd32, nearbyintd64, and nearbyintd128 Subroutines

Purpose

Round numbers to an integer value in floating-point format.

Syntax

```
#include <math.h>

double nearbyint (x)
double x;

float nearbyintf (x)
float x;

long double nearbyintl (x)
long double x;

_Decimal32 nearbyintd32(x)
_Decimal32 x;

_Decimal64 nearbyintd64(x)
_Decimal64 x;

_Decimal128 nearbyintd128(x)
_Decimal128 x;
```

Description

The **nearbyint**, **nearbyintf**, **nearbyintl**, **nearbyintd32**, **nearbyintd64**, and **nearbyintd128** subroutines round the *x* parameter to an integer value in floating-point format, using the current rounding direction and without raising the inexact floating-point exception.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **nearbyint**, **nearbyintf**, **nearbyintl**, **nearbyintd32**, **nearbyintd64**, and **nearbyintd128** subroutines return the rounded integer value.

If *x* is NaN, a NaN is returned.

If *x* is ±0, ±0 is returned.

If *x* is ±Inf, *x* is returned.

If the correct value would cause overflow, a range error occurs and the **nearbyint**, **nearbyintf**, **nearbyintl**, **nearbyintd32**, **nearbyintd64**, and **nearbyintd128** subroutines return the value of the macro **±HUGE_VAL**, **±HUGE_VALF**, **±HUGE_VALL**, **±HUGE_VAL_D32**, **±HUGE_VAL_D64**, **±HUGE_VAL_D128** (with the same sign as *x*), respectively.

nextafterd32, nextafterd64, nextafterd128, nexttowardd32, nexttowardd64, and nexttowardd128 Subroutines

Purpose

Compute the next representable decimal floating-point number.

Syntax

```
#include <math.h>
_Decimal32 nextafterd32 (x, y)
_Decimal32 x;
_Decimal32 y;

_Decimal64 nextafterd64 (x, y)
_Decimal64 x;
_Decimal64 y;

_Decimal128 nextafterd128 (x, y)
_Decimal128 x;
_Decimal128 y;

_Decimal32 nexttowardd32 (x, y)
_Decimal32 x;
_Decimal128 y;

_Decimal64 nexttowardd64 (x, y)
_Decimal64 x;
_Decimal128 y;

_Decimal128 nexttowardd128 (x, y)
_Decimal128 x;
_Decimal128 y;
```

Description

The **nextafterd32**, **nextafterd64**, and **nextafterd128** subroutines compute the next representable decimal floating-point value following the *x* value in the direction of the *y* value. Therefore, if the *y* value is less than the *x* value, the **nextafter** subroutine returns the largest representable decimal floating-point number that is less than *x*.

If the value of *x* equals *y*, the **nextafterd32**, **nextafterd64**, and **nextafterd128** subroutines return the value of *y* .

The **nexttowardd32**, **nexttowardd64**, and **nexttowardd128** subroutines are equivalent to the corresponding **nextafter** subroutines, except that the second parameter has the **_Decimal128** type, and the subroutines return the value of the *y* parameter that is converted to the type of the subroutine if the value of *x* equals that of *y*.

To check error situations, the application must set the **errno** global variable to zero and call the **feclearexcept** subroutine (**FE_ALL_EXCEPT**) before calling these subroutines. On return, if the **errno** is of the value of nonzero or the **fetestexcept** subroutine (**FE_INVALID**| **FE_DIVBYZERO**| **FE_OVERFLOW**| **FE_UNDERFLOW**) is of the value of nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the starting values. The next representable decimal floating-point number is found from the *x* parameter in the direction specified by the *y* parameter.
y	Specifies the direction.

Return Values

Upon successful completion, the **nextafterd32**, **nextafterd64**, **nextafterd128**, **nexttowardd32**, **nexttowardd64**, and **nexttowardd128** subroutines return the next representable decimal floating-point value following the value of the *x* parameter in the direction specified by the *y* parameter.

If *x* == *y*, *y* (of the *x* type) is returned.

If *x* is finite and the correct function value overflows, a range error occurs. The **±HUGE_VAL_D32**, **±HUGE_VAL_D64**, and **±HUGE_VAL_D128** (with the same sign as the *x* parameter) is returned respectively according to the returned type of the function.

If *x* or *y* is NaN, a NaN is returned.

If *x* != *y* and the correct subroutine value is subnormal, zero, or underflow, a range error occurs and either the correct function value (if representable) or a value of `0.0` is returned.

Errors

If the value of the *x* parameter is finite and the correct function value overflows, a range error occurs. The **±HUGE_VAL_D32**, **±HUGE_VAL_D64**, and **±HUGE_VAL_D128** (with the same sign as the *x* parameter) is returned respectively according to the returned type of the function.

If the value of the *x* parameter is not equal to that of the *y* parameter, and the correct subroutine value is subnormal, zero, or underflow, a range error occurs and either the correct function value (if representable) or a value of `0.0` is returned.

nextafter, nextafterf, nextafterl, nexttoward, nexttowardf, or nexttowardl Subroutine

Purpose

Computes the next representable floating-point number.

Syntax

```
#include <math.h>

float nextafterf (x, y)
float x;
float y;

long double nextafterl (x, y)
long double x;
long double y;

double nextafter (x, y)
double x, y;

double nexttoward (x, y)
double x;
long double y;

float nexttowardf (x, y)
float x;
long double y;

long double nexttowardl (x, y)
long double x;
long double y;
```

Description

The **nextafterf**, **nextafterl**, and **nextafter** subroutines compute the next representable floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, the **nextafter** subroutine returns the largest representable floating-point number less than *x*.

The **nextafter**, **nextafterf**, and **nextafterl** subroutines return *y* if *x* equals *y*.

The **nexttoward**, **nexttowardf**, and **nexttowardl** subroutines are equivalent to the corresponding **nextafter** subroutine, except that the second parameter has type **long double** and the subroutines return *y* converted to the type of the subroutine if *x* equals *y*.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the starting value. The next representable floating-point number is found from *x* in the direction specified by *y*.
y	Specifies the direction.

Return Values

Upon successful completion, the **nextafterf**, **nextafterl**, **nextafter**, **nexttoward**, **nexttowardf**, and **nexttowardl** subroutines return the next representable floating-point value following *x* in the direction of *y*.

If *x*==*y*, *y* (of the type *x*) is returned.

If *x* is finite and the correct function value would overflow, a range error occurs and ±**HUGE_VAL**, ±**HUGE_VALF**, and ±**HUGE_VALL** (with the same sign as *x*) is returned as appropriate for the return type of the function.

If *x* or *y* is NaN, a NaN is returned.

If *x*!=*y* and the correct subroutine value is subnormal, zero, or underflows, a range error occurs, and either the correct function value (if representable) or 0.0 is returned.

Error Codes

For the **nextafter** subroutine, if the *x* parameter is finite and the correct function value would overflow, **HUGE_VAL** is returned and **errno** is set to **ERANGE**.

newlocale Subroutine

Purpose

Creates or modifies a locale object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <locale.h>
```

```
locale_t newlocale(category_mask, locale, base);
int category_mask;
const char *locale;
locale_t base;
```

Description

The **newlocale** subroutine creates a new locale object or modifies an existing one. If the base argument is **(locale_t)0**, a new locale object is created.

The *category_mask* argument specifies the locale categories to be set or modified. Values for *category_mask* are constructed by a bitwise-inclusive OR of the symbolic constants LC_CTYPE_MASK , LC_NUMERIC_MASK , LC_TIME_MASK , LC_COLLATE_MASK , LC_MONETARY_MASK , and LC_MESSAGES_MASK.

For each category with the corresponding bit set in *category_mask*, the data from the locale named by the *locale* argument is used. When modifying an existing locale object, the data from the locale named by *locale* replaces the existing data within the locale object. If a completely new locale object is created, the data for all sections not requested by *category_mask* are taken from the default locale.

Special Values

The following are the special values for the *locale* parameter:

Item	Description
POSIX	Specifies the minimal environment for C-language translation called the POSIX locale.
C	Equivalent to POSIX.
""	Specifies an implementation-defined native environment. This corresponds to the value of the associated environment variables, LC_* and LANG. Refer to XBD Locale and Environment Variables.

The results are undefined if the base argument is the special locale object LC_GLOBAL_LOCALE.

Return Values

If successful, the **newlocale** subroutine returns a handle which the caller may use on subsequent calls to the **duplocale**, **freelocale**, and other subroutines that take a *locale_t* argument.

If there is failure, the **newlocale** subroutine returns **(locale_t)0** and sets the **errno** global variable to indicate the error.

Error Codes

The **newlocale** subroutine fails if the following is true:

Item	Description
ENOMEM	There is not enough memory available to create the locale object or load the locale data.
EINVAL	The *category_mask* argument contains a bit that does not correspond to a valid category.
ENOENT	For any of the categories in *category_mask* argument, the locale data is not available.

The **newlocale** subroutine may fail if the following is true:

Item	Description
EINVAL	The *locale* argument is not a valid string pointer.

Example

The following example shows the construction of a locale where the LC_CTYPE category data comes from a locale *loc1* and the LC_TIME category data from a locale *loc2*:

```
#include <locale.h>

...
locale_t loc, new_loc;
/* Get the "loc1" data. */

loc =  newlocale (LC_CTYPE_MASK, "loc1", NULL);
if (loc == (locale_t)0)
abort();
/* Get the "loc2" data. */

new_loc =  newlocale (LC_TIME_MASK, "loc2", loc);
if (new_loc != (locale_t)0)
/* We do not abort if this fails. In this case this
       simply used to unchanged locale object. */
loc = new_loc;
....
```

newpad, pnoutrefresh, prefresh, or subpad Subroutine

Purpose

Pad management functions.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

WINDOW *newpad
(int nlines,
int ncols);

int
pnoutrefresh
(WINDOW *pad,
int pminrow,
int pmincol,
int sminrow,
int smincol,
int smaxrorw,
int smaxcol);

int
prefresh
(WINDOW *pad,
int pminrow,
int pmincol,
int sminrow,
int smincol,
int smaxrorw,
int smaxcol);

WINDOW
*subpad
(WINDOW *orig,
int nlines,
int ncols,
```

```
int begin_y,
int begin_x);
```

Description

The **newpad** subroutine creates a specialised WINDOW data structure with *nlines* lines and *ncols* columns. A pad is similar to a window, except that it is not associated with a viewable part of the screen. Automatic refreshes of pads do not occur.

The **subpad** subroutine creates a subwindow within a pad with n*lines* lines and *ncols* columns. Unlike the **subwin** subroutine, which uses screen coordinates, the window is at a position (begin_y, begin_x) on the pad. The window is made in the middle of the window *orig*, so that changes made to one window affects both windows.

The **prefresh** or **pnoutrefresh** subroutines are analogous to the **wrefresh** and **wnoutrefresh** subroutines except that they relate to pads instead of windows. The additional arguments indicate what part of the pad and screen are involved. The *pminrow* and *pmincol* arguments specify the origin of the rectangle to be displayed in the screen. The lower right-hand corner of the rectangle to be displayed in the pad is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures. Negative values of *pminrow*, *pmincol*, *sminrow* or *smincol* are treated as if they were zero.

Parameters

Item	Description
ncols	
nlines	
begin_x	
begin_y	
**orig*	
**pad*	
pminrow	
pmincol	
sminrow	
smincol	
smaxrorw	
smaxcol	

Return Values

Upon successful completion, the **newpad** and **subpad** subroutines return a pointer to the pad structure. Otherwise, they return a null pointer.

Upon successful completion, the **pnoutrefresh** and **prefresh** subroutines return OK. Otherwise, they return ERR.

Examples

For the **newpad** subroutine:

1. To create a new pad and save the pointer to it in `my_pad`, enter:

```
WINDOW *my_pad;

my_pad = newpad(5, 10);
```

my_pad is now a pad 5 lines deep, 10 columns wide.

2. To create a pad and save the pointer to it in my_pad, which is flush with the right side of the terminal, enter:

```
WINDOW *my_pad;

my_pad = newpad(5, 0);
```

my_pad is now a pad 5 lines deep, extending to the far right side of the terminal.

3. To create a pad and save the pointer to it in my_pad, which fills the entire terminal, enter:

```
WINDOW *my_pad;

my_pad = newpad(0, 0);
```

my_pad is now a pad that fills the entire terminal.

4. To create a very large pad and display part of it on the screen, enter;

```
WINDOW *my_pad;

my_pal = newpad(120,120);

prefresh (my_pal, 0,0,0,0,20,30);
```

This causes the first 21 rows and first 31 columns of the pad to be displayed on the screen. The upper left coordinates of the resulting rectangle are (0,0) and the bottom right coordinates are (20,30).

For the **prefresh** or **pnoutrefresh** subroutines:

1. To update the user-defined my_pad pad from the upper-left corner of the pad on the terminal with the upper-left corner at the coordinates Y=20, X=10 and the lower-right corner at the coordinates Y=30, X=25 enter

```
WINDOW *my_pad;
prefresh(my_pad, 0, 0, 20, 10, 30, 25);
```

2. To update the user-defined my_pad1 and my_pad2 pads and output them both to the terminal in one burst of output, enter:

```
WINDOW *my_pad1; *my_pad2;
pnoutrefresh(my_pad1, 0, 0, 20, 10, 30, 25);
pnoutrefresh(my_pad2, 0, 0, 0, 0, 10, 5);
doupdate();
```

For the **subpad** subroutine:

To create a subpad, use:

```
WINDOW *orig, *mypad;
orig = newpad(100, 200);
mypad = subpad(orig, 30, 5, 25, 180);
```

The parent pad is 100 lines by 200 columns. The subpad is 30 lines by 5 columns and starts in line 25, column 180 of the parent pad.

newpass Subroutine

Purpose

Generates a new password for a user.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <userpw.h>
```

char *newpass(*Password***)**
struct userpw ***Password*;**

Description

Note: This subroutine has been depreciated and its use is not recommended. The chpass subroutine should be used in its place.

The **newpass** subroutine generates a new password for the user specified by the *Password* parameter. This subroutine displays a dialogue to enter and confirm the user's new password.

Passwords can contain almost any legal value for a character but cannot contain (National Language Support (NLS) code points. Passwords cannot have more than the value specified by **MAX_PASS**.

If a password is successfully generated, a pointer to a buffer containing the new password is returned and the last update time is reset.

Note: The **newpass** subroutine is not safe in a multithreaded environment. To use **newpass** in a threaded application, the application must keep the integrity of each thread.

Parameters

Item	Description
Password	Specifies a user password structure. This structure is defined in the **userpw.h** file and contains the following members: **upw_name** A pointer to a character buffer containing the user name. **upw_passwd** A pointer to a character buffer containing the current password. **upw_lastupdate** The time the password was last changed, in seconds since the epoch. **upw_flags** A bit mask containing 0 or more of the following values: **PW_ADMIN** This bit indicates that password information for this user may only be changed by the root user. **PW_ADMCHG** This bit indicates that the password is being changed by root and the password will have to be changed upon the next successful running of the **login** or **su** commands to this account.

Security

Item	Description
Policy: Authentication	To change a password, the invoker must be properly authenticated.

Note: Programs that invoke the **newpass** subroutine should be written to conform to the authentication rules enforced by **newpass**. The **PW_ADMCHG** flag should always be explicitly cleared unless the invoker of the command is an administrator.

Return Values

If a new password is successfully generated, a pointer to the new encrypted password is returned. If an error occurs, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **newpass** subroutine fails if one or more of the following are true:

Item	Description
EINVAL	The structure passed to the **newpass** subroutine is invalid.
ESAD	Security authentication is denied for the invoker.
EPERM	The user is unable to change the password of a user with the **PW_ADMCHG** bit set, and the real user ID of the process is not the root user.
ENOENT	The user is not properly defined in the database.

Implementation Specifics

This subroutine is part of Base Operating System (BOS) Runtime.

newpassx Subroutine

Purpose

Generates a new password for a user (without a name length limit).

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <userpw.h>

char *newpassx (Password)
struct userpwx *Password;
```

Description

Note: The **newpassx** subroutine has been obsoleted by the more current **chpassx** subroutine. Use the **chpassx** subroutine instead.

The **newpassx** subroutine generates a new password for the user specified by the *Password* parameter. The new password is then checked to ensure that it meets the password rules on the system unless the

user is exempted from these restrictions. Users must have root user authority to invoke this subroutine. The password rules are defined in the **/etc/security/user** file or the administrative domain for the user and are described in both the user file and the **passwd** command.

Passwords can contain almost any legal value for a character but cannot contain National Language Support (NLS) code points. Passwords cannot have more characters than the value specified by **PASS_MAX**.

The **newpassx** subroutine authenticates the user prior to returning the new password. If the **PW_ADMCHG** flag is set in the **upw_flags** member of the *Password* parameter, the supplied password is checked against the calling user's password. This is done to authenticate the user corresponding to the real user ID of the process instead of the user specified by the **upw_name** member of the *Password* parameter structure.

If a password is successfully generated, a pointer to a buffer containing the new password is returned and the last update time is set to the current system time. The password value in the **/etc/security/passwd** file or user's administrative domain is not modified.

Note: The **newpassx** subroutine is not safe in a multithreaded environment. To use **newpassx** in a threaded application, the application must keep the integrity of each thread.

Parameters

Item	Description
Password	Specifies a user password structure.

The fields in a **userpwx** structure are defined in the **userpw.h** file, and they include the following members:

Item	Description
upw_name	Specifies the user's name.
upw_passwd	Specifies the user's encrypted password.
upw_lastupdate	Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, 1 January 1970), when the password was last updated.
upw_flags	Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the **userpw.h** file: **PW_NOCHECK** Specifies that new passwords need not meet password restrictions in effect for the system. **PW_ADMCHG** Specifies that the password was last set by an administrator and must be changed at the next successful use of the login or **su** command. **PW_ADMIN** Specifies that password information for this user can only be changed by the root user.
upw_authdb	Specifies the administrative domain containing the authentication data.

Security

Item	Description
Policy: Authentication	To change a password, the invoker must be properly authenticated.

Note: Programs that invoke the **newpassx** subroutine should be written to conform to the authentication rules enforced by **newpassx**. The **PW_ADMCHG** flag should always be explicitly cleared unless the invoker of the command is an administrator.

Return Values

If a new password is successfully generated, a pointer to the new encrypted password is returned. If an error occurs, a null pointer is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **newpassx** subroutine fails if one or more of the following is true:

Item	Description
EINVAL	The structure passed to the **newpassx** subroutine is invalid.
ENOENT	The user is not properly defined in the database.
EPERM	The user is unable to change the password of a user with the **PW_ADMCHG** bit set, and the real user ID of the process is not the root user.
ESAD	Security authentication is denied for the invoker.

newterm Subroutine

Purpose

Initializes curses and its data structures for a specified terminal.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

SCREEN *newterm(
 Type,
 OutFile,  InFile)
char *Type;
FILE *OutFile, *InFile;
```

Description

The **newterm** subroutine initializes curses and its data structures for a specified terminal. Use this subroutine instead of the **initscr** subroutine if you are writing a program that sends output to more than one terminal. You should also use this subroutine if your program requires indication of error conditions so that it can run in a line-oriented mode on terminals that do not support a screen-oriented program.

If you are directing your program's output to more than one terminal, you must call the **newterm** subroutine once for each terminal. You must also call the **endwin** subroutine for each terminal to stop curses and restore the terminal to its previous state.

Parameters

Item	Description
InFile	Identifies the input device file.
OutFile	Identifies the output device file.
Type	Specifies the type of output terminal. This parameter is the same as the **$TERM** environment variable for that terminal.

Return Values

The **newterm** subroutine returns a variable of type **SCREEN ***. You should save this reference to the terminal within your program.

Examples

1. To initialize curses on a terminal represented by the lft device file as both the input and output terminal, open the device file with the following:

   ```
   fdfile = fopen("/dev/lft0", "r+");
   ```

   Then, use the **newterm** subroutine to initialize curses on the terminal and save the new terminal in the *my_terminal* variable as follows:

   ```
   char termname [] = "terminaltype";
   SCREEN *my_terminal;
   my_terminal = newterm(termname,fdfile, fdfile);
   ```

2. To open the device file `/dev/lft0` as the input terminal and the `/dev/tty0` (an ibm3151) as the output terminal, do the following:

   ```
   fdifile = fopen("/dev/lft0", "r");
   fdofile = fopen("/dev/tty0", "w");

   SCREEN *my_terminal2;
   my_terminal2 = newterm("ibm3151",fdofile, fdifile);
   ```

3. To use stdin for input and stdout for output, do the following:

   ```
   char termname [] = "terminaltype";
   SCREEN *my_terminal;
   my_terminal = newterm(termname,stdout,stdin);
   ```

nftw or nftw64 Subroutine

Purpose

Walks a file tree.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ftw.h>

int nftw ( Path, Function, Depth, Flags)
const char *Path;
int *(*Function) ( );
```

```
int Depth;
int Flags;
```

```
int nftw64(Path,Function,Depth)
const char *Path;
int *(*Function) ( );
int Depth;
int Flags;
```

Description

The **nftw** and **nftw64** subroutines recursively descend the directory hierarchy rooted in the *Path* parameter. The nftw and nftw64 subroutines have a similar effect to ftw and ftw64 except that they take an additional argument flags, which is a bitwise inclusive-OR of zero or more of the following flags:

Item	Description
FTW_CHDIR	If set, the current working directory will change to each directory as files are reported. If clear, the current working directory will not change.
FTW_DEPTH	If set, all files in a directory will be reported before the directory itself. If clear, the directory will be reported before any files.
FTW_MOUNT	If set, symbolic links will not be followed. If clear the links will be followed.
FTW_PHYS	If set, symbolic links will not be followed. If clear the links will be followed, and will not report the same file more than once.

For each file in the hierarchy, the **nftw** and **nftw64** subroutines call the function specified by the *Function* parameter. The nftw subroutine passes a pointer to a null-terminated character string containing the name of the file, a pointer to a stat structure containing information about the file, an integer and a pointer to an FTW structure. The nftw64 subroutine passes a pointer to a null-terminated character string containing the name of the file, a pointer to a stat64 structure containing information about the file, an integer and a pointer to an FTW structure.

The nftw subroutine uses the stat system call which will fail on files of size larger than 2 Gigabytes. The nftw64 subroutine must be used if there is a possibility of files of size larger than 2 Gigabytes.

The integer passed to the *Function* parameter identifies the file type with one of the following values:

Item	Description
FTW_F	Regular file
FTW_D	Directory
FTW_DNR	Directory that cannot be read
FTW_DP	The *Object* is a directory and subdirectories have been visited. (This condition will only occur if **FTW_DEPTH** is included in flags).
FTW_SL	Symbolic Link
FTW_SLN	Symbolic Link that does not name an existin file. (This condition will only occur if the **FTW_PHYS** flag is not included in flags).
FTW_NS	File for which the **stat** structure could not be executed successfully

If the integer is **FTW_DNR**, the files and subdirectories contained in that directory are not processed.

If the integer is **FTW_NS**, the **stat** structure contents are meaningless. An example of a file that causes **FTW_NS** to be passed to the *Function* parameter is a file in a directory for which you have read permission but not execute (search) permission.

The **FTW** structure pointer passed to the *Function* parameter contains base which is the offset of the object's filename in the pathname passed as the first argument to *Function*. The value of level indicates depth relative to the root of the walk.

The **nftw** and **nftw64** subroutines use one file descriptor for each level in the tree. The *Depth* parameter specifies the maximum number of file descriptors to be used. In general, the **nftw** and **nftw64** run faster of the value of the *Depth* parameter is at least as large as the number of levels in the tree. However, the value of the *Depth* parameter must not be greater than the number of file descriptors currently available for use. If the value of the *Depth* parameter is 0 or a negative number, the effect is the same as if it were 1.

Because the **nftw** and **nftw64** subroutines are recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **nftw** and **nftw64** subroutines use the **malloc** subroutine to allocate dynamic storage during its operation. If the **nftw** subroutine is terminated prior to its completion, such as by the **longjmp** subroutine being executed by the function specified by the *Function* parameter or by an interrupt routine, the **nftw** subroutine cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the *Function* parameter return a nonzero value the next time it is called.

Parameters

Item	Description
Path	Specifies the directory hierarchy to be searched.
Function	User supplied function that is called for each file encountered.
Depth	Specifies the maximum number of file descriptors to be used. *Depth* cannot be greater than OPEN_MAX which is described in the sys/limits.h header file.

Return Values

If the tree is exhausted, the **nftw** and **nftw64** subroutine returns a value of 0. If the subroutine pointed to by **fn** returns a nonzero value, **nftw** and **nftw64** stops its tree traversal and returns whatever value was returned by the subroutine pointed to by **fn**. If the **nftw** and **nftw64** subroutine detects an error, it returns a -1 and sets the **errno** global variable to indicate the error.

Error Codes

If the **nftw** or **nftw64** subroutines detect an error, a value of -1 is returned and the **errno** global variable is set to indicate the error.

The **nftw** and **nftw64** subroutine are unsuccessful if:

Item	Description
EACCES	Search permission is denied for any component of the *Path* parameter or read permission is denied for *Path*.
ENAMETOOLONG	The length of the path exceeds **PATH_MAX** while **_POSIX_NO_TRUNC** is in effect.
ENOENT	The *Path* parameter points to the name of a file that does not exist or points to an empty string.
ENOTDIR	A component of the *Path* parameter is not a directory.

The **nftw** subroutine is unsuccessful if:

Item	Description
EOVERFLOW	A file in *Path* is of a size larger than 2 Gigabytes.

nl or nonl Subroutine

Purpose

Enables/disables newline translation.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int nl(void);
```

```
int nonl(void);
```

Description

The **nl** subroutine enables a mode in which carriage return is translated to newline on input. The **nonnl** subroutine disables the above translation. Initially, the above translation is enabled.

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To instruct **wgetch** to translate the carriage return into a newline, enter:

   ```
   nl();
   ```

2. To instruct **wgetch** not to translate the carriage return, enter:

   ```
   nonl();
   ```

nl_langinfo Subroutine

Purpose

Returns information on the language or cultural area in a program's locale.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <nl_types.h>
#include <langinfo.h>
```

```
char *nl_langinfo ( Item)
nl_item Item;
```

Description

The **nl_langinfo** subroutine returns a pointer to a string containing information relevant to the particular language or cultural area defined in the program's locale and corresponding to the *Item* parameter. The active language or cultural area is determined by the default value of the environment variables or by the most recent call to the **setlocale** subroutine. If the **setlocale** subroutine has not been called in the program, then the default C locale values will be returned from **nl_langinfo**.

Values for the *Item* parameter are defined in the **langinfo.h** file.

The following table summarizes the categories for which **nl_langinfo()** returns information, the values the *Item* parameter can take, and descriptions of the returned strings. In the table, radix character refers to the character that separates whole and fractional numeric or monetary quantities. For example, a period (.) is used as the radix character in the U.S., and a comma (,) is used as the radix character in France.

Category	**Value of *item***	**Returned Result**
LC_MONETARY	CRNCYSTR	Currency symbol and its position.
LC_NUMERIC	RADIXCHAR	Radix character.
LC_NUMERIC	THOUSEP	Separator for the thousands.
LC_MESSAGES	YESSTR	Affirmative response for yes/no queries.
LC_MESSAGES	NOSTR	Negative response for yes/no queries.
LC_TIME	D_T_FMT	String for formatting date and time.
LC_TIME	D_FMT	String for formatting date.
LC_TIME	T_FMT	String for formatting time.
LC_TIME	AM_STR	Antemeridian affix.
LC_TIME	PM_STR	Postmeridian affix.
LC_TIME	DAY_1 through DAY_7	Name of the first day of the week to the seventh day of the week.
LC_TIME	ABDAY_1 through ABDAY-7	Abbreviated name of the first day of the week to the seventh day of the week.
LC_TIME	MON_1 through MON_12	Name of the first month of the year to the twelfth month of the year.
LC_TIME	ABMON_1 through ABMON_12	Abbreviated name of the first month of the year to the twelfth month.
LC_CTYPE	CODESET	Code set currently in use in the program.

Note: The information returned by the **nl_langinfo** subroutine is located in a static buffer. The contents of this buffer are overwritten in subsequent calls to the **nl_langinfo** subroutine. Therefore, you should save the returned information.

Parameter

Item	Description
Item	Information needed from locale.

Return Values

In a locale where language information data is not defined, the **nl_langinfo** subroutine returns a pointer to the corresponding string in the C locale. In all locales, the **nl_langinfo** subroutine returns a pointer to an empty string if the *Item* parameter contains an invalid setting.

The **nl_langinfo** subroutine returns a pointer to a static area. Subsequent calls to the **nl_langinfo** subroutine overwrite the results of a previous call.

nlist, nlist64 Subroutine

Purpose

Gets entries from a name list.

Library

Standard C Library (**libc.a**)

Berkeley Compatibility Library [**libbsd.a**] for the **nlist** subroutine, 32-bit programs, and POWER® processor-based platform

Syntax

```
#include <nlist.h>

int nlist ( FileName, NL )
const char *FileName;
struct nlist *NL;

int nlist64 ( FileName, NL64 )
const char *FileName;
struct nlist64 *NL64;
```

Description

The **nlist** and **nlist64** subroutines examine the name list in the object file named by the *FileName* parameter. The subroutine selectively reads a list of values and stores them into an array of **nlist** or **nlist64** structures pointed to by the *NL* or *NL64* parameter, respectively.

The name list specified by the *NL* or *NL64* parameter consists of an array of **nlist** or **nlist64** structures containing symbol names and other information. The list is terminated with an element that has a null pointer or a pointer to a null string in the **n_name** structure member. Each symbol name is looked up in the name list of the file. If the name is found, the value of the symbol is stored in the structure, and the other fields are filled in. If the program was not compiled with the **-g** flag, the **n_type** field may be 0.

If multiple instances of a symbol are found, the information about the last instance is stored. If a symbol is not found, all structure fields except the **n_name** field are set to 0. Only global symbols will be found.

The **nlist** and **nlist64** subroutines run in both 32-bit and 64-bit programs that read the name list of both 32-bit and 64-bit object files, with one exception: in 32-bit programs, **nlist** will return -1 if the specified file is a 64-bit object file.

The **nlist** and **nlist64** subroutines are used to read the name list from XCOFF object files.

The **nlist64** subroutine can be used to examine the system name list kept in the kernel, by specifying **/unix** as the *FileName* parameter. The **knlist** subroutine can also be used to look up symbols in the current kernel namespace.

Note: The **nlist.h** header file has a *#define* field for **n_name**. If a source file includes both **nlist.h** and **netdb.h**, there will be a conflict with the use of **n_name**. If **netdb.h** is included after **nlist.h**, **n_name** will be undefined. To correct this problem, **_n._n_name** should be used instead. If **netdb.h** is included before **nlist.h**, and you need to refer to the **n_name** field of *struct netent*, you should undefine **n_name** by entering:

```
#undef n_name
```

The **nlist** subroutine in **libbsd.a** is supported only in 32-bit mode.

Parameters

Item	Description
FileName	Specifies the name of the file containing a name list.
NL	Points to the array of **nlist** structures.
NL64	Points to the array of **nlist64** structures.

Return Values

Upon successful completion, a `0` is returned, even if some symbols could not be found. In the **libbsd.a** version of **nlist**, the number of symbols not found in the object file's name list is returned. If the file cannot be found or if it is not a valid name list, a value of `-1` is returned.

Compatibility Interfaces

To obtain the BSD-compatible version of the subroutine 32-bit applications, compile with the **libbsd.a** library by using the **-lbsd** flag.

nodelay Subroutine

Purpose

Enables or disables block during read.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int nodelay(WINDOW *win,
bool bf);
```

Description

The **nodelay** subroutine specifies whether Delay Mode or No Delay Mode is in effect for the screen associated with the specified window. If *bf* is TRUE, this screen is set to No Delay Mode. If *bf* is FALSE, this screen is set to Delay Mode. The initial state is FALSE.

Parameters

Item	Description
bf	
**win*	

Return Values

Upon successful completion, the **nodelay** subroutine returns OK. Otherwise, it returns ERR.

Examples

1. To cause the **wgetch** subroutine to return an error message, if no input is ready in the user-defined window `my_window`, use:

```
nodelay(my_window, TRUE);
```

2. To allow for a delay when retrieving a character in the user-defined window `my_window`, use:

```
WINDOW *my_window;
nodelay(my_window, FALSE);
```

notimeout, timeout, wtimeout Subroutine

Purpose

Controls blocking on input.

Library

Curses Library (**libcurses.a**)

Curses Syntax

```
#include <curses.h>

int notimeout
(WINDOW *win,
bool bf);

void timeout
(int delay);

void wtimeout
(WINDOW *win,
int delay);
```

Description

The **notimeout** subroutine specifies whether Timeout Mode or No Timeout Mode is in effect for the screen associated with the specified window. If *bf* is TRUE, this screen is set to No Timeout Mode. If *bf* is FALSE, this screen is set to Timeout Mode. The initial state is FALSE.

The **timeout** and **wtimeout** subroutines set blocking or non-blocking read for the current or specified window based on the value of delay:

Item	Description
delay < 0	One or more blocking reads (indefinite waits for input) are used.

Item	Description
delay = 0	One or more non-blocking reads are used. Any Curses input subroutine will fail if every character of the requested string is not immediately available.
delay > 0	Any Curses input subroutine blocks for delay milliseconds and fails if there is still no input.

Parameters

Item	Description
**win*	
bf	

Return Values

Upon successful completion, the **notimeout** subroutine returns OK. Otherwise, it returns ERR.

The **timeout** and **wtimeout** subroutines do not return a value.

Examples

To set the flag so that the **wgetch** subroutine does not set the timer when getting characters from the `my_win` window, use:

```
WINDOW *my_win;
notimeout(my_win, TRUE);
```

ns_addr Subroutine

Purpose

Address conversion routines.

Library

Standard C Library (**libc.a**)

Syntax

#include <sys/types.h> #include <netns/ns.h>

struct ns_addr(char ****cp*)**

Description

The **ns_addr** subroutine interprets character strings representing addresses, returning binary information suitable for use in system calls.

The **ns_addr** subroutine separates an address into one to three fields using a single delimiter and examines each field for byte separators (colon or period). The delimiters are:

Item	Description
.	period
:	colon

Item	Description
#	pound sign.

If byte separators are found, each subfield separated is taken to be a small hexadecimal number, and the entirety is taken as a network-byte-ordered quantity to be zero extended in the high-networked-order bytes. Next, the field is inspected for hyphens, which would indicate the field is a number in decimal notation with hyphens separating the millenia. The field is assumed to be a number, interpreted as hexadecimal, if a leading *0x* (as in C), a trailing *H*, (as in Mesa), or any super-octal digits are present. The field is interpreted as octal if a leading *0* is present and there are no super-octal digits. Otherwise, the field is converted as a decimal number.

Parameter

Item	Description
cp	Returns a pointer to the address of a **ns_addr** structure.

ns_ntoa Subroutine

Purpose

Address conversion routines.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <netns/ns.h>

char *ns_ntoa (
struct ns_addr ns)
```

Description

The **ns_ntoa** subroutine takes addresses and returns ASCII strings representing the address in a notation in common use in the Xerox Development Environment:

```
<network number> <host number> <port number>
```

Trailing zero fields are suppressed, and each number is printed in hexadecimal, in a format suitable for input to the **ns_addr** subroutine. Any fields lacking super-decimal digits will have a trailing *H* appended.

Note: The string returned by **ns_ntoa** resides in static memory.

Parameter

Item	Description
ns	Returns a pointer to a string.

ntimeradd Macro

Purpose

Computes the sum of nanotimers.

Syntax

```
#include<sys/time.h>
ntimeradd(tvp, svp, rvp)
struct timestruc_t tvp, svp, rvp;
```

Note: The `ntimeradd` macro asserts for invalid values of parameters. The following header files need to be included for definition of assert:

<assert.h>
For user applications.

<sys/syspest.h>
For kernel extensions.

Description

The ntimeradd macro is used to compute the sum of nanotimers. It adds the nanotimer values that are stored in the *tvp* and *svp* variables and stores the result in the *rvp* variable.

Parameters

The *tvp* , *svp*, and *rvp* variables are of type `struct timestruc_t` structure, that is defined in the `sys/time.h` header file.

ntimersub Macro

Purpose

Computes the difference between two nanotimers.

Syntax

```
#include<sys/time.h>
ntimersub(tvp, svp, rvp)
struct timestruc_t tvp, svp, rvp;
```

Note: The `ntimersub` macro asserts for invalid values of parameters. The following header files need to be included for definition of assert:

<assert.h>
For user applications.

<sys/syspest.h>
For kernel extensions.

Description

The `ntimersub` macro is used to compute the difference between nanotimers. Call to the `ntimersub(tvp, svp, rvp)` macro subtracts the nanotimer value that is stored in the *svp* variable from the value that is stored in the *tvp* variable and stores the result in the *rvp* variable.

Parameters

The *tvp* , *svp*, and *rvp* variables are of type `struct timestruc_t` structure, that is defined in the `sys/time.h` header file.

o

The following Base Operating System (BOS) runtime services begin with the letter *o*.

odm_add_obj Subroutine

Purpose

Adds a new object into an object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_add_obj ( ClassSymbol,  DataStructure)
CLASS_SYMBOL ClassSymbol;
struct ClassName *DataStructure;
```

Description

The **odm_add_obj** subroutine takes as input the class symbol that identifies both the object class to add and a pointer to the data structure containing the object to be added.

The **odm_add_obj** subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

Parameters

Item	Description
ClassSymbol	Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this identifier is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.
DataStructure	Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the *ClassSymbol* parameter. The structure is declared in the **.h** file created by the **odmcreate** command and has the same name as the object class.

Return Values

Upon successful completion, an identifier for the object that was added is returned. If the **odm_add_obj** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_add_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**

- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_PARAMS**
- **ODMI_READ_ONLY**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_change_obj Subroutine

Purpose

Changes an object in the object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_change_obj ( ClassSymbol,  DataStructure)
CLASS_SYMBOL ClassSymbol;
struct ClassName *DataStructure;
```

Description

The **odm_change_obj** subroutine takes as input the class symbol that identifies both the object class to change and a pointer to the data structure containing the object to be changed. The application program must first retrieve the object with an **odm_get_obj** subroutine call, change the data values in the returned structure, and then pass that structure to the **odm_change_obj** subroutine.

The **odm_change_obj** subroutine opens and closes the object class around the change if the object class was not previously opened. If the object class was previously opened, then the subroutine leaves the object class open when it returns.

Parameters

Item	Description
ClassSymbol	Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this identifier is the *ClassName*_**CLASS** structure that is created by the **odmcreate** command.
DataStructure	Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the *ClassSymbol* parameter. The structure is declared in the **.h** file created by the **odmcreate** command and has the same name as the object class.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_change_obj** subroutine fails, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_change_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_NO_OBJECT**
- **ODMI_OPEN_ERR**
- **ODMI_PARAMS**
- **ODMI_READ_ONLY**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_close_class Subroutine

Purpose

Closes an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

int odm_close_class (*ClassSymbol*)
CLASS_SYMBOL *ClassSymbol*;

Description

The **odm_close_class** subroutine closes the specified object class.

Parameters

Item	Description
ClassSymbol	Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this identifier is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_close_class** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_close_class** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_DNE**

- **ODMI_CLASS_PERMS**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_create_class Subroutine

Purpose

Creates an object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_create_class ( ClassSymbol)
CLASS_SYMBOL ClassSymbol;
```

Description

The **odm_create_class** subroutine creates an object class. However, the **.c** and **.h** files generated by the **odmcreate** command are required to be part of the application.

Parameters

Item	Description
ClassSymbol	Specifies a class symbol of the form *ClassName*_**CLASS,** which is declared in the **.h** file created by the **odmcreate** command.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_create_class** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_create_class** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_EXISTS**
- **ODMI_CLASS_PERMS**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_OPEN_ERR**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_err_msg Subroutine

Purpose

Returns an error message string.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
int odm_err_msg ( ODMErrno,  MessageString)
long ODMErrno;
char **MessageString;
```

Description

The **odm_err_msg** subroutine takes as input an *ODMErrno* parameter and an address in which to put the string pointer of the message string that corresponds to the input ODM error number. If no corresponding message is found for the input error number, a null string is returned and the subroutine is unsuccessful.

Parameters

Item	Description
ODMErrno	Specifies the error code for which the message string is retrieved.
MessageString	Specifies the address of a string pointer that will point to the returned error message string.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_err_msg** subroutine is unsuccessful, a value of -1 is returned, and the *MessageString* value returned is a null string.

Examples

The following example shows the use of the **odm_err_msg** subroutine:

```
#include <odmi.h>
char *error_message;

...
/*-------------------------------------------------------------*/
/*ODMErrno was returned from a previous ODM subroutine call.*/
/*-------------------------------------------------------------*/
returnstatus = odm_err_msg ( odmerrno, &error_message );
if ( returnstatus < 0 )
   printf ( "Retrieval of error message failed\n" );
else
   printf ( error_message );
```

odm_free_list Subroutine

Purpose

Frees memory previously allocated for an **odm_get_list** subroutine.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
int odm_free_list ( ReturnData,  DataInfo)
struct ClassName *ReturnData;
struct listinfo *DataInfo;
```

Description

The **odm_free_list** subroutine recursively frees up a tree of memory object lists that were allocated for an **odm_get_list** subroutine.

Parameters

Item	Description
ReturnData	Points to the array of *ClassName* structures returned from the **odm_get_list** subroutine.
DataInfo	Points to the **listinfo** structure that was returned from the **odm_get_list** subroutine. The **listinfo** structure has the following form:

```
struct listinfo {
char ClassName[16];      /* class name for query */
char criteria[256];      /* query criteria */
int num;                 /* number of matches found */
int valid;               /* for ODM use */
CLASS_SYMBOL class;      /* symbol for queried class */
};
```

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_free_list** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_free_list** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_MAGICNO_ERR**
- **ODMI_PARAMS**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_get_by_id Subroutine

Purpose

Retrieves an object from an ODM object class by its ID.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
struct ClassName *odm_get_by_id( ClassSymbol,  ObjectID,  ReturnData)
CLASS_SYMBOL ClassSymbol;
int ObjectID;
struct ClassName *ReturnData;
```

Description

The **odm_get_by_id** subroutine retrieves an object from an object class. The object to be retrieved is specified by passing its *ObjectID* parameter from its corresponding *ClassName* structure.

Parameters

Item	Description
ClassSymbol	Specifies a class symbol identifier of the form *ClassName*_**CLASS,** which is declared in the **.h** file created by the **odmcreate** command.
ObjectID	Specifies an identifier retrieved from the corresponding *ClassName* structure of the object class.
ReturnData	Specifies a pointer to an instance of the C language structure corresponding to the object class referenced by the *ClassSymbol* parameter. The structure is declared in the **.h** file created by the **odmcreate** command and has the same name as the object class.

Return Values

Upon successful completion, a pointer to the *ClassName* structure containing the object is returned. If the **odm_get_by_id** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_by_id** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_MALLOC_ERR**
- **ODMI_NO_OBJECT**

- **ODMI_OPEN_ERR**
- **ODMI_PARAMS**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_get_list Subroutine

Purpose

Retrieves all objects in an object class that match the specified criteria.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
struct ClassName *odm_get_list (ClassSymbol, Criteria, ListInfo, MaxReturn, LinkDepth)
struct ClassName_CLASS  ClassSymbol; char * Criteria; struct listinfo *  ListInfo;
int  MaxReturn,   LinkDepth;
```

Description

The **odm_get_list** subroutine takes an object class and criteria as input, and returns a list of objects that satisfy the input criteria. The subroutine opens and closes the object class around the subroutine if the object class was not previously opened. If the object class was previously opened, the subroutine leaves the object class open when it returns.

Parameters

Item	Description
ClassSymbol	Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this is the *ClassName*_**CLASS** structure created by the **odmcreate** command.
Criteria	Specifies a string that contains the qualifying criteria for selecting the objects to remove.

ListInfo	Specifies a structure containing information about the retrieval of the objects. The **listinfo** structure has the following form:
	`struct listinfo {`
	`char ClassName[16];  /* class name used for query */`
	`char criteria[256];  /* query criteria */`
	`int num;  /* number of matches found */`
	`int valid;  /* for ODM use */`
	`CLASS_SYMBOL class;  /* symbol for queried class */`
	`};`

MaxReturn	Specifies the expected number of objects to be returned. This is used to control the increments in which storage for structures is allocated, to reduce the **realloc** subroutine copy overhead.

Item	Description
LinkDepth	Specifies the number of levels to recurse for objects with **ODM_LINK** descriptors. A setting of 1 indicates only the top level is retrieved; 2 indicates **ODM_LINK**s will be followed from the top/first level only: 3 indicates **ODM_LINK**s will be followed at the first and second level, and so on.

Return Values

Upon successful completion, a pointer to an array of C language structures containing the objects is returned. This structure matches that described in the **.h** file that is returned from the **odmcreate** command. If no match is found, null is returned. If the **odm_get_list** subroutine fails, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_list** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_BAD_CRIT**
- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_INTERNAL_ERR**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_LINK_NOT_FOUND**
- **ODMI_MAGICNO_ERR**
- **ODMI_MALLOC_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_PARAMS**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_get_obj, odm_get_first, or odm_get_next Subroutine

Purpose

Retrieves objects, one object at a time, from an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
struct ClassName *odm_get_obj ( ClassSymbol, Criteria, ReturnData, FIRST_NEXT)
```

```
struct ClassName *odm_get_first (ClassSymbol, Criteria, ReturnData)
```

```
struct ClassName *odm_get_next (ClassSymbol, ReturnData)
```

```
CLASS_SYMBOL ClassSymbol;
char *Criteria;
struct ClassName *ReturnData;
int FIRST_NEXT;
```

Description

The **odm_get_obj**, **odm_get_first**, and **odm_get_next** subroutines retrieve objects from ODM object classes and return the objects into C language structures defined by the **.h** file produced by the **odmcreate** command.

The **odm_get_obj**, **odm_get_first**, and **odm_get_next** subroutines open and close the specified object class if the object class was not previously opened. If the object class was previously opened, the subroutines leave the object class open upon return.

Parameters

Item	Description
ClassSymbol	Specifies a class symbol identifier returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, then this identifier is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.
Criteria	Specifies the string that contains the qualifying criteria for retrieval of the objects.
ReturnData	Specifies the pointer to the data structure in the **.h** file created by the **odmcreate** command. The name of the structure in the **.h** file is *ClassName*. If the *ReturnData* parameter is null (`ReturnData == null`), space is allocated for the parameter and the calling application is responsible for freeing this space at a later time. If variable length character strings (vchar) are returned, they are referenced by pointers in the *ReturnData* structure. Calling applications must free each vchar between each call to the **odm_get** subroutines; otherwise storage will be lost.
FIRST_NEXT	Specifies whether to get the first object that matches the criteria or the next object. Valid values are: **ODM_FIRST** Retrieve the first object that matches the search criteria. **ODM_NEXT** Retrieve the next object that matches the search criteria. The *Criteria* parameter is ignored if the *FIRST_NEXT* parameter is set to **ODM_NEXT**.

Return Values

Upon successful completion, a pointer to the retrieved object is returned. If no match is found, null is returned. If an **odm_get_obj**, **odm_get_first**, or **odm_get_next** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_get_obj**, **odm_get_first** or **odm_get_next** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_BAD_CRIT**
- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_INTERNAL_ERR**
- **ODMI_INVALID_CLXN**

- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_MALLOC_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_initialize Subroutine

Purpose

Prepares ODM for use by an application.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
int odm_initialize( )
```

Description

The **odm_initialize** subroutine starts ODM for use with an application program.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_initialize** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_initialize** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_INVALID_PATH**
- **ODMI_MALLOC_ERR**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_lock Subroutine

Purpose

Puts an exclusive lock on the requested path name.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
int odm_lock ( LockPath,  TimeOut)
char *LockPath;
int TimeOut;
```

Description

The **odm_lock** subroutine is used by an application to prevent other applications or methods from accessing an object class or group of object classes. A lock on a directory path name does not prevent another application from acquiring a lock on a subdirectory or object class within that directory.

Note: Coordination of locking is the responsibility of the application accessing the object classes.

The **odm_lock** subroutine returns a lock identifier that is used to call the **odm_unlock** subroutine.

Parameters

Item	Description
LockPath	Specifies a string containing the path name in the file system in which to locate object classes or the path name to an object class to lock.
TimeOut	Specifies the amount of time, in seconds, to wait if another application or method holds a lock on the requested object class or classes. The possible values for the *TimeOut* parameter are: ***TimeOut* = ODM_NOWAIT** The **odm_lock** subroutine is unsuccessful if the lock cannot be granted immediately. ***TimeOut* = *Integer*** The **odm_lock** subroutine waits the specified amount of seconds to retry an unsuccessful lock request. ***TimeOut* = ODM_WAIT** The **odm_lock** subroutine waits until the locked path name is freed from its current lock and then locks it.

Return Values

Upon successful completion, a lock identifier is returned. If the **odm_lock** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_lock** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_BAD_LOCK**
- **ODMI_BAD_TIMEOUT**
- **ODMI_BAD_TOKEN**
- **ODMI_LOCK_BLOCKED**
- **ODMI_LOCK_ENV**
- **ODMI_MALLOC_ERR**
- **ODMI_UNLOCK**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_mount_class Subroutine

Purpose

Retrieves the class symbol structure for the specified object class name.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

CLASS_SYMBOL odm_mount_class (*ClassName*)
char **ClassName*;

Description

The **odm_mount_class** subroutine retrieves the class symbol structure for a specified object class. The subroutine can be called by applications (for example, the ODM commands) that have no previous knowledge of the structure of an object class before trying to access that class. The **odm_mount_class** subroutine determines the class description from the object class header information and creates a **CLASS_SYMBOL** object class that is returned to the caller.

The object class is not opened by the **odm_mount_class** subroutine. Calling the subroutine subsequent times for an object class that is already open or mounted returns the same **CLASS_SYMBOL** object class.

Mounting a class that links to another object class recursively mounts to the linked class. However, if the recursive mount is unsuccessful, the original **odm_mount_class** subroutine does not fail; the **CLASS_SYMBOL** object class is set up with a null link.

Parameters

Item	Description
ClassName	Specifies the name of an object class from which to retrieve the class description.

Return Values

Upon successful completion, a **CLASS_SYMBOL** is returned. If the **odm_mount_class** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_mount_class** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_BAD_CLASSNAME**
- **ODMI_BAD_CLXNNAME**
- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_CLXNMAGICNO_ERR**
- **ODMI_INVALID_CLASS**
- **ODMI_INVALID_CLXN**
- **ODMI_MAGICNO_ERR**

- **ODMI_MALLOC_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_PARAMS**
- **ODMI_TOOMANYCLASSES**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_open_class or odm_open_class_rdonly Subroutine

Purpose

Opens an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

CLASS_SYMBOL odm_open_class ( ClassSymbol)
CLASS_SYMBOL ClassSymbol;

CLASS_SYMBOL odm_open_class_rdonly ( ClassSymbol)
CLASS_SYMBOL ClassSymbol;
```

Description

The **odm_open_class** subroutine can be called to open an object class. Most subroutines implicitly open a class if the class is not already open. However, an application may find it useful to perform an explicit open if, for example, several operations must be done on one object class before closing the class. The **odm_open_class_rdonly** subroutine opens an **odm** database in read-only mode.

Parameter

Item	Description
ClassSymbol	Specifies a class symbol of the form *ClassName*_**CLASS** that is declared in the **.h** file created by the **odmcreate** command.

Return Values

Upon successful completion, a *ClassSymbol* parameter for the object class is returned. If the **odm_open_class** or **odm_open_class_rdonly** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_open_class** or **odm_open_class_rdonly** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_INVALID_PATH**

- **ODMI_MAGICNO_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_rm_by_id Subroutine

Purpose

Removes objects specified by their IDs from an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
int odm_rm_by_id( ClassSymbol,  ObjectID)
CLASS_SYMBOL ClassSymbol;
int ObjectID;
```

Description

The **odm_rm_by_id** subroutine is called to delete an object from an object class. The object to be deleted is specified by passing its object ID from its corresponding *ClassName* structure.

Parameters

Item	Description
ClassSymbol	Identifies a class symbol returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, this is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.
ObjectID	Identifies the object. This information is retrieved from the corresponding *ClassName* structure of the object class.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_rm_by_id** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_by_id** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_FORK**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_MALLOC_ERR**

- **ODMI_NO_OBJECT**
- **ODMI_OPEN_ERR**
- **ODMI_OPEN_PIPE**
- **ODMI_PARAMS**
- **ODMI_READ_ONLY**
- **ODMI_READ_PIPE**
- **ODMI_TOOMANYCLASSES**
- **ODMI_TOOMANYCLASSES**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_rm_class Subroutine

Purpose

Removes an object class from the file system.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_rm_class ( ClassSymbol)
CLASS_SYMBOL ClassSymbol;
```

Description

The **odm_rm_class** subroutine removes an object class from the file system. All objects in the specified class are deleted.

Parameter

Item	Description
ClassSymbol	Identifies a class symbol returned from the **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, this is the *ClassName*_**CLASS** structure created by the **odmcreate** command.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_rm_class** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_class** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**

- **ODMI_MAGICNO_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_TOOMANYCLASSES**
- **ODMI_UNLINKCLASS_ERR**
- **ODMI_UNLINKCLXN_ERR**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_rm_obj Subroutine

Purpose

Removes objects from an ODM object class.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_rm_obj ( ClassSymbol,  Criteria)
CLASS_SYMBOL ClassSymbol;
char *Criteria;
```

Description

The **odm_rm_obj** subroutine deletes objects from an object class.

Parameters

Item	Description
ClassSymbol	Identifies a class symbol returned from an **odm_open_class** subroutine. If the **odm_open_class** subroutine has not been called, this is the *ClassName*_**CLASS** structure that was created by the **odmcreate** command.
Criteria	Contains as a string the qualifying criteria for selecting the objects to remove.

Return Values

Upon successful completion, the number of objects deleted is returned. If the **odm_rm_obj** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_rm_obj** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_BAD_CRIT**
- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_FORK**
- **ODMI_INTERNAL_ERR**
- **ODMI_INVALID_CLXN**

- **ODMI_INVALID_PATH**
- **ODMI_MAGICNO_ERR**
- **ODMI_MALLOC_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_OPEN_PIPE**
- **ODMI_PARAMS**
- **ODMI_READ_ONLY**
- **ODMI_READ_PIPE**
- **ODMI_TOOMANYCLASSES**

odm_run_method Subroutine

Purpose

Runs a specified method.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

int odm_run_method(*MethodName*, *MethodParameters*, *NewStdOut*, *NewStdError*)
char * *MethodName*, * *MethodParameters*;
char ** *NewStdOut*, ** *NewStdError*;

Description

The **odm_run_method** subroutine takes as input the name of the method to run, any parameters for the method, and addresses of locations for the **odm_run_method** subroutine to store pointers to the stdout (standard output) and stderr (standard error output) buffers. The application uses the pointers to access the stdout and stderr information generated by the method.

Parameters

Item	Description
MethodName	Indicates the method to execute. The method can already be known by the applications, or can be retrieved as part of an **odm_get_obj** subroutine call.
MethodParameters	Specifies a list of parameters for the specified method.
NewStdOut	Specifies the address of a pointer to the memory where the standard output of the method is stored. If the *NewStdOut* parameter is a null value (`NewStdOut == NULL`), standard output is not captured.
NewStdError	Specifies the address of a pointer to the memory where the standard error output of the method will be stored. If the *NewStdError* parameter is a null value (`NewStdError == NULL`), standard error output is not captured.

Return Values

If successful, the **odm_run_method** subroutine returns the exit status and *out_ptr* and *err_ptr* should contain the relevant information. If unsuccessful, the **odm_run_method** subroutine will return -1 and set the **odmerrno** variable to an error code.

Note: AIXmethods usually return the exit code defined in the **cf.h** file if the methods exit on error.

Error Codes

Failure of the **odm_run_method** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_FORK**
- **ODMI_MALLOC_ERR**
- **ODMI_OPEN_PIPE**
- **ODMI_PARAMS**
- **ODMI_READ_PIPE**

odm_set_path Subroutine

Purpose

Sets the default path for locating object classes.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

char *odm_set_path ( NewPath)
char *NewPath;
```

Description

The **odm_set_path** subroutine is used to set the default path for locating object classes. The subroutine allocates memory, sets the default path, and returns the pointer to memory. Once the operation is complete, the calling application should free the pointer using the **free** (../m_bostechref/malloc.dita) subroutine.

Parameters

Item	Description
NewPath	Contains, as a string, the path name in the file system in which to locate object classes.

Return Values

Upon successful completion, a string pointing to the previous default path is returned. If the **odm_set_path** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_set_path** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_INVALID_PATH**
- **ODMI_MALLOC_ERR**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_set_perms Subroutine

Purpose

Sets the default permissions for an ODM object class at creation time.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
int odm_set_perms ( NewPermissions)
int NewPermissions;
```

Description

The **odm_set_perms** subroutine defines the default permissions to assign to object classes at creation.

Parameters

Item	Description
NewPermissions	Specifies the new default permissions parameter as an integer.

Return Values

Upon successful completion, the current default permissions are returned. If the **odm_set_perms** subroutine is unsuccessful, a value of -1 is returned.

odm_terminate Subroutine

Purpose

Terminates an ODM session.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>
```

```
int odm_terminate ( )
```

Description

The **odm_terminate** subroutine performs the cleanup necessary to terminate an ODM session. After running an **odm_terminate** subroutine, an application must issue an **odm_initialize** subroutine to resume ODM operations.

Return Values

Upon successful completion, a value of 0 is returned. If the **odm_terminate** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_terminate** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_CLASS_DNE**
- **ODMI_CLASS_PERMS**
- **ODMI_INVALID_CLXN**
- **ODMI_INVALID_PATH**
- **ODMI_LOCK_ID**
- **ODMI_MAGICNO_ERR**
- **ODMI_OPEN_ERR**
- **ODMI_TOOMANYCLASSES**
- **ODMI_UNLOCK**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

odm_unlock Subroutine

Purpose

Releases a lock put on a path name.

Library

Object Data Manager Library (**libodm.a**)

Syntax

```
#include <odmi.h>

int odm_unlock ( LockID)
int LockID;
```

Description

The **odm_unlock** subroutine releases a previously granted lock on a path name. This path name can be a directory containing subdirectories and object classes.

Parameters

Item	Description
LockID	Identifies the lock returned from the **odm_lock** subroutine.

Return Values

Upon successful completion a value of 0 is returned. If the **odm_unlock** subroutine is unsuccessful, a value of -1 is returned and the **odmerrno** variable is set to an error code.

Error Codes

Failure of the **odm_unlock** subroutine sets the **odmerrno** variable to one of the following error codes:

- **ODMI_LOCK_ID**
- **ODMI_UNLOCK**

See ../bostechref/odm_error_codes.dita for explanations of the ODM error codes.

open, openat, openx, openxat, open64, open64at, open64x, open64xat, creat, or creat64 Subroutine

Purpose

Opens a file for reading or writing.

Syntax

```
#include <fcntl.h>

int open (Path, OFlag [, Mode])
const char *Path;
int OFlag;
mode_t Mode;

int openat (DirFileDescriptor, Path, OFlag [, Mode])
int DirFileDescriptor;
const char *Path;
int OFlag;
mode_t Mode;

int openx (Path, OFlag, Mode, Extension)
const char *Path;
int OFlag;
mode_t Mode;
long Extension;

int openxat (DirFileDescriptor, Path, OFlag, Mode, Extension)
int DirFileDescriptor;
const char * Path;
int OFlag;
mode_t Mode;
long Extension;

int creat (Path, Mode)
const char *Path;
mode_t Mode;

int open64 (Path, OFlag [, Mode])
const char *Path;
int OFlag;
mode_t Mode;

int open64at (DirFileDescriptor, Path,
OFlag [, Mode])
```

```
int DirFileDescriptor;
const char * Path;
int OFlag;
mode_t Mode;
```

```
int creat64 (Path, Mode)
const char *Path;
mode_t Mode;
```

```
int open64x (Path, OFlag, Mode, Extension)
char *Path;
int64_t OFlag;
mode_t Mode;
ext_t Extension;
```

```
int open64xat (DirFileDescriptor, Path, OFlag, Mode, Extension)
int DirFileDescriptor;
char *Path;
int64_t OFlag;
mode_t Mode;
ext_t Extension;
```

Description

The **openat** subroutine is equivalent to the **open** subroutine if the *DirFileDescriptor* parameter is **AT_FDCWD** or the *Path* parameter is an absolute path name. If *DirFileDescriptor* is a valid file descriptor of an open directory and *Path* is a relative path name, *Path* is considered to be relative to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory. Similarly, the **openxat**, **open64at**, or **open64xat** subroutine are equivalent to the **openx**, **open64**, or **open64x** subroutine, respectively, in the same way as **openat** and **open**.

The **open**, **openx**, and **creat** subroutines establish a connection between the file named by the *Path* parameter and a file descriptor. The opened file descriptor is used by subsequent I/O subroutines, such as **read** and **write**, to access that file.

The **openx** subroutine is the same as the **open** subroutine, with the addition of an *Extension* parameter, which is provided for device driver use. The **creat** subroutine is equivalent to the **open** subroutine with the **O_WRONLY**, **O_CREAT**, and **O_TRUNC** flags set.

The returned file descriptor is the lowest file descriptor not previously open for that process. No process can have more than **OPEN_MAX** file descriptors open simultaneously.

The file offset, marking the current position within the file, is set to the beginning of the file. The new file descriptor is set to remain open across exec subroutines.

The **open64** and **creat64** subroutines are equivalent to the **open** and **creat** subroutines except that the **O_LARGEFILE** flag is set in the open file description associated with the returned file descriptor. This flag allows files larger than **OFF_MAX** to be accessed. If the caller attempts to open a file larger than **OFF_MAX** and **O_LARGEFILE** is not set, the open will fail and **errno** will be set to **EOVERFLOW**.

In the large file enabled programming environment, **open** is redefined to be **open64** and **creat** is redefined to be **creat64**.

The **open64x** subroutine creates and accesses an encrypted file in an Encrypting File System (EFS). The **open64x** subroutine is similar to the **openx** subroutine, with the modification of the *OFlag* parameter, which is updated to a 64-bit quantity.

If the *DirFileDescriptor* parameter in the **openat**, **openxat**, **open64at**, or **open64xat** subroutine was opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted for that directory by using the current permissions of the directory. If the directory was opened with the **O_SEARCH** open flag, the subroutine does not perform the check for that directory.

Parameters

Item	Description
DirFileDescriptor	Specifies the file descriptor of an open directory.
Path	Specifies the file to be opened. If *DirFileDescriptor* is specified and Path is a relative path name, then *Path* is considered relative to the directory specified by *DirFileDescriptor*.

Item	Description
Mode	Specifies the read, write, and execute permissions of the file to be created (requested by the **O_CREAT** flag). If the file already exists, this parameter is ignored. The *Mode* parameter is constructed by logically ORing one or more of the following values, which are defined in the **<sys/mode.h>** file: **S_ISUID** Enables the **setuid** attribute for an executable file. A process executing this program acquires the access rights of the owner of the file. **S_ISGID** Enables the **setgid** attribute for an executable file. A process executing this program acquires the access rights of the group of the file. Also, enables the group-inheritance attribute for a directory. Files created in this directory have a group equal to the group of the directory. The following attributes apply only to files that are directly executable. They have no meaning when applied to executable text files such as shell scripts and **awk** scripts. **S_ISVTX** Enables the **link/unlink** attribute for a directory. Files cannot be linked to in this directory. Files can only be unlinked if the requesting process has write permission for the directory and is either the owner of the file or the directory. **S_ISVTX** Enables the **save text** attribute for an executable file. The program is not unmapped after usage. **S_ENFMT** Enables enforcement-mode record locking for a regular file. File locks requested with the **lockf** subroutine are enforced. **S_IRUSR** Permits the file's owner to read it. **S_IWUSR** Permits the file's owner to write to it. **S_IXUSR** Permits the file's owner to execute it (or to search the directory). **S_IRGRP** Permits the file's group to read it. **S_IWGRP** Permits the file's group to write to it. **S_IXGRP** Permits the file's group to execute it (or to search the directory). **S_IROTH** Permits others to read the file. **S_IWOTH** Permits others to write to the file. **S_IXOTH** Permits others to execute the file (or to search the directory). Other mode values exist that can be set with the **mknod** subroutine but not with the **chmod** subroutine.

Item	Description
Extension	Provides communication with character device drivers that require additional information or return additional status. Each driver interprets the *Extension* parameter in a device-dependent way, either as a value or as a pointer to a communication area. Drivers must apply reasonable defaults when the *Extension* parameter value is 0.
OFlag	Specifies the type of access, special open processing, the type of update, and the initial state of the open file. The parameter value is constructed by logically ORing special open processing flags. These flags are defined in the **fcntl.h** file and are described in the following flags.

Flags That Specify Access Type

The following *OFlag* parameter flag values specify type of access:

Item	Description
O_RDONLY	The file is opened for reading only.
O_WRONLY	The file is opened for writing only.
O_RDWR	The file is opened for both reading and writing.
O_SEARCH	The directory is opened for search only. If the *Path* parameter does not point to an existing directory, the flag is ignored.

Note: One of the file access values must be specified. Do not use **O_RDONLY**, **O_WRONLY**, or **O_RDWR** together. If none is set, none is used, and the results are unpredictable.

Flags That Specify Special Open Processing

The following *OFlag* parameter flag values specify special open processing:

Item	Description
O_CREAT	If the file exists, this flag has no effect, except as noted under the **O_EXCL** flag. If the file does not exist, a regular file is created with the following characteristics: • The owner ID of the file is set to the effective user ID of the process. • The group ID of the file is set to the group ID of the parent directory if the parent directory has the **SetGroupID** attribute (**S_ISGID** bit) set. Otherwise, the group ID of the file is set to the effective group ID of the calling process. • The file permission and attribute bits are set to the value of the *Mode* parameter, modified as follows: – All bits set in the process file mode creation mask are cleared. (The file creation mask is described in the **umask** subroutine.) – The **S_ISVTX** attribute bit is cleared. The file open with the O_CREAT flag by the open64 subroutine must create an encrypted file when the file is within an encrypted directory or inheritance schema and the calling process has an open key store. This will have the effect of generating a random symmetric file encryption key, wrapping it with the user's public key and storing it in the file's metadata.
O_EFSON	Along with the **O_CREAT** flag, this flag explicitly creates an encrypted file in a file-system that is EFS enabled, overriding inheritance. This function is available for the open64x subroutine.
O_EFSOFF	Along with the **O_CREAT** flag, this flag explicitly overrides inheritance to create a non-encrypted file. This function is available for the open64x subroutine.

Item	Description
O_DIRECTORY	The subroutine is unsuccessful if the *Path* parameter does not point to a directory.
O_EXCL	If the **O_EXCL** and **O_CREAT** flags are set, the open is unsuccessful if the file exists. **Note:** The **O_EXCL** flag is not fully supported for Network File Systems (NFS). The NFS protocol does not guarantee the designed function of the **O_EXCL** flag.
O_NSHARE	Assures that no process has this file open and precludes subsequent opens. If the file is on a physical file system and is already open, this open is unsuccessful and returns immediately unless the *OFlag* parameter also specifies the **O_DELAY** flag. This flag is effective only with physical file systems. **Note:** This flag is not supported by NFS.
O_RSHARE	Assures that no process has this file open for writing and precludes subsequent opens for writing. The calling process can request write access. If the file is on a physical file system and is open for writing or open with the **O_NSHARE** flag, this open fails and returns immediately unless the *OFlag* parameter also specifies the **O_DELAY** flag. **Note:** This flag is not supported by NFS.
O_RAW	To read or write the encrypted file in raw-mode without holding the encryption key. This function is available for the open64x subroutine.
O_DEFER	The file is opened for deferred update. Changes to the file are not reflected on permanent storage until an **fsync** subroutine operation is performed. If no **fsync** subroutine operation is performed, the changes are discarded when the file is closed. **Note:** This flag is not supported by NFS or JFS2, and the flag will be quietly ignored. **Note:** This flag causes modified pages to be backed by paging space. Before using this flag make sure there is sufficient paging space.
O_NOCTTY	This flag specifies that the controlling terminal should not be assigned during this open.
O_TRUNC	If the file does not exist, this flag has no effect. If the file exists, is a regular file, and is successfully opened with the **O_RDWR** flag or the **O_WRONLY** flag, all of the following apply: • The length of the file is truncated to 0. • The owner and group of the file are unchanged. • The **SetUserID** attribute of the file mode is cleared. • The **SetUserID** attribute of the file is cleared.
O_DIRECT	This flag specifies that direct i/o will be used for this file while it is opened.

Item	Description
O_CIO	This flag specifies that concurrent i/o (CIO) will be used for the file while it is opened. Because implementing concurrent readers and writers utilizes the direct I/O path (with more specific requirements to improve performance for running database on the file system), this flag will override the **O_DIRECT** flag if the two options are specified at the same time. The length of data to be read or written and the file offset must be page-aligned to be transferred as direct i/o with concurrent readers and writers. The **O_CIO** flag is exclusive. If the file is opened in any other way (for example, using the **O_DIRECT** flag or opening the file normally), the open will fail. If the file is opened using the **O_CIO** flag and another process to open the file another way, the open will fail. The **O_CIO** flag also prevents the **mmap** subroutine and the **shmat** subroutine access to the file. The **mmap** subroutine and the **shmat** subroutine return **EINVAL** if they are used on a file that was opened using the **O_CIO** flag.
O_CIOR	This flag specifies that concurrent I/O will be used for the file while it is opened. This flag can only be used in conjuction with **O_CIO**. In addition this flag also specifies that another process can open the file in read-only mode. All the other ways to open the file will fail. This flag is only available with the **open64x ()** interface. The other varieties of open allow only flags defined in the low-order 32 bits.
O_SNAPSHOT	The file being opened contains a JFS2 snapshot. Subsequent **read** calls using this file descriptor will read the cooked snapshot rather than the raw snapshot blocks. A snapshot can only have one active open file descriptor for it. The **O_SNAPSHOT** option is available only for external snapshot.

The **open** subroutine is unsuccessful if any of the following conditions are true:

- The file supports enforced record locks and another process has locked a portion of the file.
- The file is on a physical file system and is already open with the **O_RSHARE** flag or the **O_NSHARE** flag.
- The file does not allow write access.
- The file is already opened for deferred update.

Flag That Specifies Type of Update

A program can request some control on when updates should be made permanent for a regular file opened for write access. The following *OFlag* parameter values specify the type of update performed:

Item	Description
O_SYNC:	If set, updates to regular files and writes to block devices are synchronous updates. File update is performed by the following subroutines: • **fclear** • **ftruncate** • **open** with **O_TRUNC** • **write** On return from a subroutine that performs a synchronous update (any of the preceding subroutines, when the **O_SYNC** flag is set), the program is assured that all data for the file has been written to permanent storage, even if the file is also open for deferred update.

Item	Description
O_DSYNC:	If set, the file data as well as all file system meta-data required to retrieve the file data are written to their permanent storage locations. File attributes such as access or modification times are not required to retrieve file data, and as such, they are not guaranteed to be written to their permanent storage locations before the preceding subroutines return. (Subroutines listed in the **O_SYNC** description.)
O_SYNC | O_DSYNC:	If both flags are set, the file's data and all of the file's meta-data (including access time) are written to their permanent storage locations.

Item	Description
O_RSYNC:	This flag is used in combination with **O_SYNC** or **D_SYNC**, and it extends their write operation behaviors to read operations. For example, when **O_SYNC** and **R_SYNC** are both set, a read operation will not return until the file's data and all of the file's meta-data (including access time) are written to their permanent storage locations.

Flags That Define the Open File Initial State

The following *OFlag* parameter flag values define the initial state of the open file:

Item	Description
O_APPEND	The file pointer is set to the end of the file prior to each write operation.
O_DELAY	Specifies that if the **open** subroutine could not succeed due to an inability to grant the access on a physical file system required by the **O_RSHARE** flag or the **O_NSHARE** flag, the process blocks instead of returning the **ETXTBSY** error code.
O_NDELAY	Opens with no delay.
O_NONBLOCK	Specifies that the **open** subroutine should not block.

The **O_NDELAY** flag and the **O_NONBLOCK** flag are identical except for the value returned by the **read** and **write** subroutines. These flags mean the process does not block on the state of an object, but does block on input or output to a regular file or block device.

The **O_DELAY** flag is relevant only when used with the **O_NSHARE** or **O_RSHARE** flags. It is unrelated to the **O_NDELAY** and **O_NONBLOCK** flags.

General Notes on OFlag Parameter Flags

The effect of the **O_CREAT** flag is immediate, even if the file is opened with the **O_DEFER** flag.

When opening a file on a physical file system with the **O_NSHARE** flag or the **O_RSHARE** flag, if the file is already open with conflicting access the following can occur:

- If the **O_DELAY** flag is clear (the default), the **open** subroutine is unsuccessful.
- If the **O_DELAY** flag is set, the **open** subroutine blocks until there is no conflicting open. There is no deadlock detection for processes using the **O_DELAY** flag.

When opening a file on a physical file system that has already been opened with the **O_NSHARE** flag, the following can occur:

- If the **O_DELAY** flag is clear (the default), the open is unsuccessful immediately.
- If the **O_DELAY** flag is set, the open blocks until there is no conflicting open.

When opening a file with the **O_RDWR**, **O_WRONLY**, or **O_TRUNC** flag, and the file is already open with the **O_RSHARE** flag:

- If the **O_DELAY** flag is clear (the default), the open is unsuccessful immediately.

- If the **O_DELAY** flag is set, the open blocks until there is no conflicting open.

When opening a first-in-first-out (FIFO) with the **O_RDONLY** flag, the following can occur:

- If the **O_NDELAY** and **O_NONBLOCK** flags are clear, the open blocks until a process opens the file for writing. If the file is already open for writing (even by the calling process), the **open** subroutine returns without delay.
- If the **O_NDELAY** flag or the **O_NONBLOCK** flag is set, the open succeeds immediately even if no process has the FIFO open for writing.

When opening a FIFO with the **O_WRONLY** flag, the following can occur:

- If the **O_NDELAY** and **O_NONBLOCK** flags are clear (the default), the open blocks until a process opens the file for reading. If the file is already open for writing (even by the calling process), the **open** subroutine returns without delay.
- If the **O_NDELAY** flag or the **O_NONBLOCK** flag is set, the **open** subroutine returns an error if no process currently has the file open for reading.

When opening a block special or character special file that supports nonblocking opens, such as a terminal device, the following can occur:

- If the **O_NDELAY** and **O_NONBLOCK** flags are clear (the default), the open blocks until the device is ready or available.
- If the **O_NDELAY** flag or the **O_NONBLOCK** flag is set, the **open** subroutine returns without waiting for the device to be ready or available. Subsequent behavior of the device is device-specific.

Any additional information on the effect, if any, of the **O_NDELAY**, **O_RSHARE**, **O_NSHARE**, and **O_DELAY** flags on a specific device is documented in the description of the special file related to the device type.

If path refers to a STREAMS file, *oflag* may be constructed from **O_NONBLOCK** OR-ed with either **O_RDONLY**, **O_WRONLY** or **O_RDWR**. Other flag values are not applicable to STREAMS devices and have no effect on them. The value **O_NONBLOCK** affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of **O_NONBLOCK** is device-specific.

If path names the controller side of a pseudo-terminal device, then it is unspecified whether **open** locks the worker side so that it cannot be opened. Portable applications must call **unlockpt** before opening the worker side.

The **O_SEARCH** flag has the same value as the **O_EXEC** flag. Starting in AIX 7.1, programs that passed the **O_EXEC** flag to a directory open may fail, as the open code will also check the search permission for the directory.

The largest value that can be represented correctly in an object of type **off_t** will be established as the offset maximum in the open file description.

Return Values

Upon successful completion, the file descriptor, a nonnegative integer, is returned. Otherwise, a value of -1 is returned, no files are created or modified, and the **errno** global variable is set to indicate the error.

Error Codes

The **open**,**openat openx**, **openxat**, **open64**, **open64at**, **open64x**, **open64xat**, and **creat** subroutines are unsuccessful and the named file is not opened if one or more of the following are true:

Item	Description
EACCES	One of the following is true: • The file exists and the type of access specified by the *OFlag* parameter is denied. • Search permission is denied on a component of the path prefix specified by the *Path* parameter. Access could be denied due to a secure mount. • The file does not exist and write permission is denied for the parent directory of the file to be created. • The **O_TRUNC** flag is specified and write permission is denied.
EAGAIN	The **O_TRUNC** flag is set and the named file contains a record lock owned by another process.
EDQUOT	The directory in which the entry for the new link is being placed cannot be extended, or an i-node could not be allocated for the file, because the user or group quota of disk blocks or i-nodes in the file system containing the directory has been exhausted.
EEXIST	The **O_CREAT** and **O_EXCL** flags are set and the named file exists.
EFBIG	An attempt was made to write a file that exceeds the process' file limit or the maximum file size. If the user has set the environment variable **XPG_SUS_ENV=ON** prior to execution of the process, then the **SIGXFSZ** signal is posted to the process when exceeding the process' file size limit.
EINTR	A signal was caught during the **open** subroutine.
EIO	The *path* parameter names a STREAMS file and a hangup or error occurred.
EISDIR	Named file is a directory and write access is required (the **O_WRONLY** or **O_RDWR** flag is set in the *OFlag* parameter).
EMFILE	The system limit for open file descriptors per process has already been reached (**OPEN_MAX**).
ENAMETOOLONG	The length of the *Path* parameter exceeds the system limit (**PATH_MAX**); or a path-name component is longer than **NAME_MAX** and **_POSIX_NO_TRUNC** is in effect.
ENFILE	The system file table is full.
ENOENT	The **O_CREAT** flag is not set and the named file does not exist; or the **O_CREAT** flag is not set and either the path prefix does not exist or the *Path* parameter points to an empty string.
ENOTDIR	The **O_DIRECTORY** flag is set and the Path parameter does not point to an existing directory.
ENOMEM	The *Path* parameter names a STREAMS file and the system is unable to allocate resources.
ENOSPC	The directory or file system that would contain the new file cannot be extended.
ENOSR	The *Path* argument names a STREAMS-based file and the system is unable to allocate a STREAM.
ENOTDIR	A component of the path prefix specified by the *Path* component is not a directory.

Item	Description
ENXIO	One of the following is true: • Named file is a character special or block special file, and the device associated with this special file does not exist. • Named file is a multiplexed special file and either the channel number is outside of the valid range or no more channels are available. • The **O_DELAY** flag or the **O_NONBLOCK** flag is set, the named file is a FIFO, the **O_WRONLY** flag is set, and no process has the file open for reading.
EOVERFLOW	A file greater than one terabyte was opened on the 32-bit kernel in JFS2. The exact max size is specified in **MAX_FILESIZE** and may be obtained using the **pathconf** system call. Any file larger than that cannot be opened on the 32-bit kernel, but can be created and opened on the 64-bit kernel.
EROFS	Named file resides on a read-only file system and write access is required (either the **O_WRONLY**, **O_RDWR**, **O_CREAT** (if the file does not exist), or **O_TRUNC** flag is set in the *OFlag* parameter).
ETXTBSY	File is on a physical file system and is already open in a manner (with the **O_RSHARE** or **O_NSHARE** flag) that precludes this open; or the **O_NSHARE** or **O_RSHARE** flag was requested with the **O_NDELAY** flag set, and there is a conflicting open on a physical file system.
ENOATTR	No keystore has been loaded in this process.
ESAD	No key available in keystore for the owner of the new file.

Item	Description
EOVERFLOW	A call was made to **open** and **creat** and the file already existed and its size was larger than **OFF_MAX** and the **O_LARGEFILE** flag was not set.

The **open**, **openx**, **open64x**, and **creat** subroutines are unsuccessful if one of the following are true:

Item	Description
EFAULT	The *Path* parameter points outside of the allocated address space of the process.
EINVAL	The value of the *OFlag* parameter is not valid.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ETXTBSY	The file specified by the *Path* parameter is a pure procedure (shared text) file that is currently executing, and the **O_WRONLY** or **O_RDWR** flag is set in the *OFlag* parameter.

The **openat**, **openxat**, **open64at**, and **open64xat** subroutines are unsuccessful and the named file is not opened if one or more of the following are true:

Item	Description
EACCES	The directory pointed at by the *DirFileDescriptor* parameter was not opened with the **O_SEARCH** flag and the search permission is denied on the directory.
EBADF	The *Path* parameter does not specify an absolute path and the DirFileDescriptor parameter is neither **AT_FDCWD** nor a valid file descriptor.
ENOTDIR	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.

open_memstream, open_wmemstream Subroutines

Purpose

Open a dynamic memory buffer stream.

Library

Standard Library (**libc.a**)

Syntax

```
#include <stdio.h>
FILE *open_memstream(char **bufp, size_t *sizep);
#include <wchar.h>
FILE *open_wmemstream(wchar_t **bufp, size_t *sizep);
```

Description

The **open_memstream( )** and **open_wmemstream( )** functions create an I/O stream associated with a dynamically allocated memory buffer. The stream is opened for writing and will be retrievable.

The stream associated with a call to **open_memstream( )** is byte-oriented.

The stream associated with a call to **open_wmemstream( )** is wide-oriented.

The stream maintains a current position in the allocated buffer and a current buffer length. The position is initially set to zero (the start of the buffer). Each write to the stream will start at the current position and move this position by the number of successfully written bytes for **open_memstream( )** or the number of successfully written wide characters for **open_wmemstream( )**. The length is initially set to zero. If a write moves the position to a value larger than the current length, the current length will be set to this position. In this case a null character for **open_memstream( )** or a null wide character for **open_wmemstream( )** will be appended to the current buffer. For both functions the terminating null is not included in the calculation of the buffer length.

After a successful **fflush( )** or **fclose( )**, the pointer referenced by *bufp* contains the address of the buffer, and the variable pointed to by *sizep* contains the number of successfully written bytes for **open_memstream( )** or the number of successfully written wide characters for **open_wmemstream( )**. The buffer is terminated by a null character for **open_memstream( )** or a null wide character for **open_wmemstream( )**.

After a successful **fflush( )** the pointer referenced by **bufp** and the variable referenced by **sizep** remain valid only until the next write operation on the stream or a call to **fclose( )**.

Return Values

Upon successful completion, these functions return a pointer to the object controlling the stream. Otherwise, a null pointer is returned, and *errno* is set to indicate the error.

Error Codes

These functions might fail if:

Item	Description
[EINVAL]	*bufp* or *sizep* are NULL.
[EMFILE]	{FOPEN_MAX} streams are currently open in the calling process.
[ENOMEM]	Memory for the stream or the buffer could not be allocated.

Examples

```
#include <stdio.h>
int main (void)
{
    FILE *stream;
    char *buf;
    size_t len;
    stream = open_memstream(&buf, &len);
    if (stream == NULL)
        /* handle error */;
    fprintf(stream, "hello my world");
    fflush(stream);
    printf("buf=%s, len=%zu\n", buf, len);
    fseeko(stream, 0, SEEK_SET);
    fprintf(stream, "good-bye");
    fclose(stream);
    printf("buf=%s, len=%zu\n", buf, len);
    free(buf);
    return 0;
}
```

This program produces the following output:

```
buf=hello my world, len=14
buf=good-bye world, len=14
```

opendir, readdir, telldir, seekdir, rewinddir, closedir, opendir64, readdir64, telldir64, seekdir64, rewinddir64, closedir64, or fdopendir Subroutine

Purpose

Performs operations on directories.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <dirent.h>
```

DIR *opendir (*DirectoryName*)
const char *DirectoryName;

```
struct dirent *readdir ( DirectoryPointer)
DIR *DirectoryPointer;
```

```
long int telldir(DirectoryPointer)
DIR *DirectoryPointer;
```

```
void seekdir(DirectoryPointer,Location)
DIR *DirectoryPointer;
long Location;
```

```
void rewinddir (DirectoryPointer)
DIR *DirectoryPointer;
```

```
int closedir (DirectoryPointer)
DIR *DirectoryPointer;
```

```
DIR *opendir64 ( DirectoryName)
const char *DirectoryName;
```

```
struct dirent64 *readdir64 ( DirectoryPointer)
DIR64 *DirectoryPointer;
```

```
offset_t telldir64(DirectoryPointer)
DIR64 *DirectoryPointer;
```

```
void seekdir64(DirectoryPointer,Location)
DIR64 *DirectoryPointer;
offset_t Location;
```

```
void rewinddir64 (DirectoryPointer)
DIR64 *DirectoryPointer;
```

```
int closedir64 (DirectoryPointer)
DIR64 *DirectoryPointer;
```

```
DIR *fdopendir(fd);
int fd;
```

Description

Attention: Do not use the **readdir** subroutine in a multithreaded environment. See the multithread alternative in the **readdir_r** subroutine article.

The **opendir** subroutine opens the directory designated by the *DirectoryName* parameter and associates a directory stream with it.

Note: An open directory must always be closed with the **closedir** subroutine to ensure that the next attempt to open that directory is successful.

The **opendir** subroutine also returns a pointer to identify the directory stream in subsequent operations. The null pointer is returned when the directory named by the *DirectoryName* parameter cannot be accessed or when not enough memory is available to hold the entire stream. A successful call to any of the **exec** functions closes any directory streams opened in the calling process.

The *fdopendir()* function is equivalent to the *opendir()* function, except that the directory is specified by a file descriptor rather than by a name. The file offset associated with the file descriptor at the time of the call, determines the entries that are returned.

Upon the successful return from *fdopendir()*, the file descriptor is under the control of the system, and if any attempt is made to close the file descriptor, or to modify the state of the associated description, other than by means of *closedir()*, *readdir()*, *readdir_r()*, or *rewinddir()*, the behavior is undefined. Upon calling *closedir()* the file descriptor is closed.

The **readdir** subroutine returns a pointer to the next directory entry. The **readdir** subroutine returns entries for . (dot) and .. (dot dot), if present, but never returns an invalid entry (with d_ino set to 0). When it reaches the end of the directory, or when it detects an invalid **seekdir** operation, the **readdir** subroutine returns the null value. The returned pointer designates data that may be overwritten by another call to the **readdir** subroutine on the same directory stream. A call to the **readdir** subroutine on a different directory stream does not overwrite this data. The **readdir** subroutine marks the st_atime field of the directory for update each time the directory is actually read.

The **telldir** subroutine returns the current location associated with the specified directory stream.

The **seekdir** subroutine sets the position of the next **readdir** subroutine operation on the directory stream. An attempt to seek an invalid location causes the **readdir** subroutine to return the null value the next time it is called. The position should be that returned by a previous **telldir** subroutine call.

The **rewinddir** subroutine resets the position of the specified directory stream to the beginning of the directory.

The **closedir** subroutine closes a directory stream and frees the structure associated with the *DirectoryPointer* parameter. If the closedir subroutine is called for a directory that is already closed, the behavior is undefined. To prevent this, always initialize the *DirectoryPointer* parameter to null after closure.

If you use the **fork** subroutine to create a new process from an existing one, either the parent or the child (but not both) may continue processing the directory stream using the **readdir**, **rewinddir**, or **seekdir** subroutine.

The opendir64 subroutine is similar to the opendir subroutine except that it returns a pointer to an object of type DIR64.

Note: An open directory by opendir64 subroutine must always be closed with the closedir64 subroutine to ensure that the next attempt to open that directory is successful. In addition, it must be operated using the 64-bit interfaces (readdir64, telldir64, seekdir64, rewinddir64, and closedir64) to obtain the correct directory information.

The readdir64 subroutine is similar to the readdir subroutine except that it returns a pointer to an object of type struct dirent64.

The telldir64 subroutine is similar to the telldir subroutine except that it returns the current directory location in an offset_t format.

The seekdir64 subroutine is similar to the seekdir subroutine except that the *Location* parameter is set in the format of offset_t.

The rewinddir64 subroutine resets the position of the specified directory stream (obtained by the opendir64 subroutine) to the beginning of the directory.

Parameters

Item	Description
DirectoryName	Names the directory.
DirectoryPointer	Points to the **DIR** or DIR64 structure of an open directory.
Location	Specifies the offset of an entry relative to the start of the directory.

Return Values

On successful completion, the **opendir**, and **fdopendir** subroutines returns a pointer to an object of type **DIR**, and the opendir64 subroutine returns a pointer to an object of type DIR64. Otherwise, a null value is returned and the **errno** global variable is set to indicate the error.

On successful completion, the **readdir** subroutine returns a pointer to an object of type **struct dirent**, and the readdir64 subroutine returns a pointer to an object of type struct dirent64. Otherwise, a null

value is returned and the **errno** global variable is set to indicate the error. When the end of the directory is encountered, a null value is returned and the **errno** global variable is not changed by this function call.

On successful completion, the `telldir` or `telldir64` subroutine returns the current location associated with the specified directory stream. Otherwise, a null value is returned.

On successful completion, the **closedir** or **closedir64** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

If the **opendir** subroutine is unsuccessful, it returns a null value and sets the **errno** global variable to one of the following values:

Item	Description
EACCES	Indicates that search permission is denied for any component of the *DirectoryName* parameter, or read permission is denied for the *DirectoryName* parameter.
ENAMETOOLONG	Indicates that the length of the *DirectoryName* parameter argument exceeds the **PATH_MAX** value, or a path-name component is longer than the **NAME_MAX** value while the **POSIX_NO_TRUNC** value is in effect.
ENOENT	Indicates that the named directory does not exist.
ENOTDIR	Indicates that a component of the *DirectoryName* parameter is not a directory.
EMFILE	Indicates that too many file descriptors are currently open for the process.
ENFILE	Indicates that too many file descriptors are currently open in the system.

If the **readdir** or `readdir64` subroutine is unsuccessful, it returns a null value and sets the **errno** global variable to the following value:

Item	Description
EBADF	Indicates that the *DirectoryPointer* parameter argument does not refer to an open directory stream.

If the **closedir** or `closedir64` subroutine is unsuccessful, it returns a value of -1 and sets the **errno** global variable to the following value:

Item	Description
EBADF	Indicates that the *DirectoryPointer* parameter argument does not refer to an open directory stream.

If the **fdopendir** subroutine is unsuccessful, it returns a null value and sets the **errno** global variable to one of the following values:

Item	Description
EBADF	Indicates that the *fd* argument is not a valid file descriptor open for reading.
ENOTDIR	Indicates that the descriptor *fd* is not associated with a directory.

Examples

To search a directory for the entry `name`:

```
len = strlen(name);
DirectoryPointer = opendir(".");
for (dp = readdir(DirectoryPointer); dp != NULL; dp =
```

```
 readdir(DirectoryPointer))
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                closedir(DirectoryPointer);
                DirectoryPointer=NULL     //To prevent multiple closure
                return FOUND;
        }
closedir(DirectoryPointer);
                DirectoryPointer=NULL     //To prevent multiple closure
```

overlay or overwrite Subroutine

Purpose

Copies one window on top of another.

Library

Curses Library (**libcurses.a**)

Syntax

```
WINDOW *dstwin);
```

```
int overwrite(const WINDOW *srcwin,
WINDOW *dstwin);
```

Description

The **overlay** and **overwrite** subroutines overlay *srcwin* on top of *dstwin*. The *scrwin* and *dstwin* arguments need not be the same size; only text where the two windows overlap is copied.

The **overwrite** subroutine copies characters as though a sequence of **win_wch** and **wadd_wch** subroutines were performed with the destination window's attributes and background attributes cleared.

The **overlay** subroutine does the same thing, except that, whenever a character to be copied is the background character of the source window. the **overlay** subroutine does not copy the character but merely moves the destination cursor the width of the source background character.

If any portion of the overlaying window border is not the first column of a multi-column character then all the column positions will be replaced with the background character and rendition before the overlay is done. If the default background character is a multi-column character when this occurs, then these subroutines fail.

Parameters

Item	Description
srcwin	
deswin	

Return Values

Upon successful completion. these subroutines return OK. Otherwise, they return ERR.

Examples

1. To copy `my_window` on top of `other_window`, excluding spaces, use:

```
WINDOW *my_window, *other_window;
overlay(my_window, other_window);
```

2. To copy `my_window` on top of `other_window`, including spaces, use:

```
WINDOW *my_window, *other_window;
overwrite(my_window, other_window);
```

p

The following Base Operating System (BOS) runtime services begin with the letter *p*.

__pthread_atexit_np Subroutine

Purpose

Registers a handler routine to be invoked when the calling thread exits.

Library

Threads library (`libpthreads.a`)

Syntax

```
#include <pthread.h>

int __pt_atexit_np (flags, handler_routine, ...)
int flags;
int (*handler_routine) (int, ...);
```

Description

The **__pt_atexit_np** subroutine adds the specified handler routine to a stack of handler routines for the calling thread. When the calling thread exits by using the **pthread_exit()** subroutine, the calling thread's handler routines are removed from the stack, one at a time. These handler routines are invoked after the cleanup routines are called and after the thread-specific data is cleaned up.

The **flags** parameter must be set to 0. If a nonzero value is specified, the `EINVAL` error code is returned. The handler function contains a **flags** parameter and optional parameters. Each handler function is invoked with a single 0 argument. The return value of a handler function must be 0. Nonzero values are reserved for future use.

If a handler routine calls the **__pt_atexit_np** subroutine to register additional handler routines, the additional routines are pushed onto the stack of handler routines. These additional routines are called when the registering handler routine returns.

If a thread calls `exit()`, its handler routines are invoked before any other processing takes place, such as calling at-exit routines. In this case, handler routines in other threads are not called unless other threads are canceled by the exiting thread. If a handler calls the **pthread_exit()** subroutine, the thread exits without causing the process to exit.

If a thread calls the `exec()` function, the handler routines are not called for any thread.

If a thread calls the `fork()` function, its handler routines remain registered in the child process.

Note: You cannot remove a handler routine from the stack of registered handler routines.

Parameters

flags
The only allowed value is 0.

handler_routine
Points to the handler routine to be invoked by the thread. The handler routine is invoked with a single 0 argument. The handler routine must return 0. Nonzero values are reserved for future use.

Return values

If successful, the **__pt_atexit_np** subroutine returns 0. Otherwise, an error number is returned to indicate the error.

Error codes

The **__pt_atexit_np** subroutine fails if the following error code is returned:

EINVAL
The **flags** parameter is not 0.

pair_content Subroutine

Purpose

Returns the colors in a color pair.

Library

Curses Library (**libcurses.a**)

Curses Syntax

```
#include <curses.h>

pair_content ( Pair,  F,  B)
short Pair;
short *F, *B;
```

Description

The **pair_content** subroutine returns the colors in a color pair. A color pair is made up of a foreground and background color. You must call the **start_color** subroutine before calling the **pair_content** subroutine.

Note: The color pair must already be initialized before calling the **pair_content** subroutine.

Return Values

Item	Description
OK	Indicates the subroutine completed successfully.
ERR	Indicates the pair has not been initialized.

Parameters

Item	Description
Pair	Identifies the color-pair number. The *Pair* parameter must be between **1** and **COLORS_PAIRS-1**.
F	Points to the address where the foreground color will be stored. The *F* parameter will be between **0** and **COLORS-1**.
B	Points to the address where the background color will be stored. The *B* parameter will be between **0** and **COLORS-1**.

Example

To obtain the foreground and background colors for color-pair 5, use:

```
short *f, *b;
pair_content(5,f,b);
```

For this subroutine to succeed, you must have already initialized the color pair. The foreground and background colors will be stored at the locations pointed to by *f* and *b*.

pam_acct_mgmt Subroutine

Purpose

Validates the user's account.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_acct_mgmt (PAMHandle, Flags)
pam_handle_t *PAMHandle;
int Flags;
```

Description

The **pam_acct_mgmt** subroutine performs various checks on the user's account to determine if it is valid. These checks can include account and password expiration, and access restrictions. This subroutine is generally used subsequent to a successful **pam_authenticate()** call in order to verify whether the authenticated user should be granted access.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The Flags argument can be a logically OR'd combination of the following: • **PAM_SILENT** – No messages should be displayed • **PAM_DISALLOW_NULL_AUTHTOK** – Do not authenticate a user with a NULL authentication token.

Return Values

Upon successful completion, **pam_acct_mgmt** returns **PAM_SUCCESS**. If the routine fails, a different error will be returned, depending on the actual error.

Error Codes

Item	Description
PAM_ACCT_EXPIRED	The user's account has expired.
PAM_NEW_AUTHTOK_REQD	The user's password needs changed. This is usually due to password aging or because it was last set by an administrator. At this stage most user's can still change their passwords; applications should call **pam_chauthtok()** and have the user promptly change their password.
PAM_AUTHTOK_EXPIRED	The user's password has expired. Unlike **PAM_NEW_AUTHTOK_REQD**, the password cannot be changed by the user.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_authenticate Subroutine

Purpose

Attempts to authenticate a user through PAM.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_authenticate (PAMHandle, Flags)
pam_handle_t *PAMHandle;
int Flags;
```

Description

The **pam_authenticate** subroutine authenticates a user through PAM. The authentication method used is determined by the authentication modules configured in the **/etc/pam.conf** stack. Most authentication requires a password or other user input but is dependent on the modules in use.

Before attempting authentication through **pam_authenticate**, ensure that all of the applicable PAM information has been set through the initial call to **pam_start()** and subsequent calls to **pam_set_item()**. If any necessary information is not set, PAM modules can prompt the user for information through the routine defined in **PAM_CONV**. If required information is not provided and **PAM_CONV** is not set, the authentication fails.

On failure, it is the responsibility of the calling application to maintain a count of authentication attempts and to reinvoke the subroutine if the count has not exceeded a defined limit. Some authentication modules maintain an internal count and return **PAM_MAXTRIES** if the limit is reached. After the stack of authentication modules has finished with either success or failure, **PAM_AUTHTOK** is cleared in the handle.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The Flags argument can be a logically OR'd combination of the following: • **PAM_SILENT** – No messages should be displayed • **PAM_DISALLOW_NULL_AUTHTOK** – Do not authenticate a user with a NULL authentication token.

Return Values

Upon successful completion, **pam_authenticate** returns **PAM_SUCCESS**. If the routine fails, a different error will be returned, depending on the actual error.

Error Codes

Item	Description
PAM_AUTH_ERR	An error occurred in authentication, usually because of an invalid authentication token.
PAM_CRED_INSUFFICIENT	The user has insufficient credentials to access the authentication data.
PAM_AUTHINFO_UNAVAIL	The authentication information cannot be retrieved.
PAM_USER_UNKNOWN	The user is not known.
PAM_MAXTRIES	The maximum number of authentication retries has been reached.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_chauthtok Subroutine

Purpose

Changes the user's authentication token (typically passwords).

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_chauthtok (PAMHandle, Flags)
pam_handle_t *PAMHandle;
int Flags;
```

Description

The **pam_chauthtok** subroutine changes a user's authentication token through the PAM framework. Prior to changing the password, the subroutine performs preliminary tests to ensure that necessary hosts and information, depending on the password service, are there. If any of these tests fail, **PAM_TRY_AGAIN** is returned. To request information from the user, **pam_chauthtok** can use the conversation function that is defined in the PAM handle, *PAMHandle*. After the subroutine is finished, the values of **PAM_AUTHTOK** and **PAM_OLDAUTHTOK** are cleared in the handle for added security.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The Flags argument can be a logically OR'd combination of the following: • **PAM_SILENT** – No messages should be displayed • **PAM_CHANGE_EXPIRED_AUTHTOK** – Only expired passwords should be changed. If this flag is not included, all users using the related password service are forced to update their passwords. This is typically used by a login application after determining password expiration. It should not generally be used by applications dedicated to changing passwords.

Return Values

Upon successful completion, **pam_chauthtok** returns **PAM_SUCCESS** and the authentication token of the user, as defined for a given password service, is changed. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_AUTHTOK_ERR	A failure occurred while updating the authentication token.
PAM_TRY_AGAIN	Preliminary checks for changing the password have failed. Try again later.
PAM_AUTHTOK_RECOVERY_ERR	An error occurred while trying to recover the authentication information.

Item	Description
PAM_AUTHTOK_LOCK_BUSY	Cannot get the authentication token lock. Try again later.
PAM_AUTHTOK_DISABLE_AGING	Authentication token aging checks are disabled and were not performed.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_close_session Subroutine

Purpose

Ends a currently open PAM user session.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_close_session (PAMHandle, Flags)
pam_handle_t *PAMHandle;
int Flags;
```

Description

The **pam_close_session** subroutine ends a PAM user session started by **pam_open_session()**.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The following flag may be set: • **PAM_SILENT** – No messages should be displayed

Return Values

Upon successful completion, **pam_close_session** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_SESSION_ERR	An error occurred while creating/removing an entry for the new session.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_end Subroutine

Purpose

Ends an existing PAM authentication session.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_end (PAMHandle, Status)
pam_handle_t *PAMHandle;
int Status;
```

Description

The **pam_end** subroutine finishes and cleans up the authentication session represented by the PAM handle *PAMHandle*. *Status* denotes the current state of the *PAMHandle* and is passed through to a **cleanup()** function so that the memory used during that session can be properly unallocated. The **cleanup()** function can be set in the *PAMHandle* by PAM modules through the **pam_set_data()** routine. Upon completion of the subroutine, the PAM handle and associated memory is no longer valid.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.

Item	Description
Status	The state of the last PAM call. Some modules need to be cleaned according to error codes.

Return Values

Upon successful completion, **pam_end** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.

pam_get_data Subroutine

Purpose

Retrieves information for a specific PAM module for this PAM session.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_get_data (PAMHandle, ModuleDataName, Data)
pam_handle_t *PAMHandle;
const char *ModuleDataName;
void **Data;
```

Description

The **pam_get_data** subroutine is used to retrieve module-specific data from the PAM handle. This subroutine is used by modules and should not be called by applications. If the *ModuleDataName* identifier exists, the reference for its data is returned in *Data*. If the identifier does not exist, a NULL reference is returned in Data. The caller should not modify or free the memory returned in *Data*. Instead, a cleanup function should be specified through a call to **pam_set_data()**. The cleanup function will be called when **pam_end()** is invoked in order to free any memory allocated.

Parameters

Item	Description
PAMHandle (in)	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
ModuleDataName	A unique identifier for Data.
Data	Returned reference to the data denoted by *ModuleDataName*.

Return Values

Upon successful completion, **pam_get_data** returns **PAM_SUCCESS**. If *ModuleDataName* exists and **pam_get_data** completes successfully, *Data* will be a valid reference. Otherwise, *Data* will be NULL. If the routine fails, either **PAM_SYSTEM_ERR**, **PAM_BUF_ERR**, or **PAM_NO_MODULE_DATA** is returned, depending on the actual error.

Error Codes

Item	Description
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_NO_MODULE_DATA	No module-specific data was found.

pam_get_item Subroutine

Purpose

Retrieves an item or information for this PAM session.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_get_item (PAMHandle, ItemType, Item)
pam_handle_t *PAMHandle;
int ItemType;
void **Item;
```

Description

The **pam_get_item** subroutine returns the item requested by the *ItemType*. Any items returned by **pam_get_item** should not be modified or freed. They can be later used by PAM and will be cleaned-up by **pam_end()**. If a requested *ItemType* is not found, a NULL reference will be returned in *Item*.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.

Item	Description
ItemType	The type of item that is being requested. The following values are valid item types: • **PAM_SERVICE** – The service name requesting this PAM session. • **PAM_USER** – The user name of the user being authenticated. • **PAM_AUTHTOK** – The user's current authentication token (password). • **PAM_OLDAUTHOK** – The user's old authentication token (old password). • **PAM_TTY** – The terminal name. • **PAM_RHOST** – The name of the remote host. • **PAM_RUSER** – The name of the remote user. • **PAM_CONV** – The **pam_conv** structure for conversing with the user. • **PAM_USER_PROMPT** – The default prompt for the user (used by **pam_get_user()**). For security, **PAM_AUTHTOK** and **PAM_OLDAUTHTOK** are only available to PAM modules.
Item	The return value, holding a reference to a pointer of the requested *ItemType*.

Return Values

Upon successful completion, **pam_get_item** returns **PAM_SUCCESS**. Also, the address of a reference to the requested object is returned in *Item*. If the requested item was not found, a NULL reference is returned. If the routine fails, either **PAM_SYSTEM_ERR** or **PAM_BUF_ERR** is returned and *Item* is set to a NULL pointer.

Error Codes

Item	Description
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_SYMBOL_ERR	Symbol not found.

pam_get_user Subroutine

Purpose

Gets the user's name from the PAM handle or through prompting for input.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_get_user (
pam_handle_t *pamh,
char **user,
const char *prompt);
```

Description

The **pam_get_user** subroutine returns the user name currently stored in the PAM handle, *pamh*. If the user name has not already been set through **pam_start()** or **pam_set_item()**, the subroutine displays the string specified by *prompt*, to prompt for the user name through the conversation function. If *prompt* is NULL, the value of **PAM_USER_PROMPT** set through a call to **pam_set_item()** is used. If both *prompt* and **PAM_USER_PROMPT** are NULL, PAM defaults to use the following string:

```
Please enter user name:
```

After the user name has been retrieved, it is set in the PAM handle and is also returned to the caller in the *user* argument. The caller should not change or free *user*, as cleanup will be handled by **pam_end()**.

Parameters

Item	Description
pamh	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
user	The user name retrieved from the PAM handle or provided by the user.
prompt	The prompt to be displayed if a user name is required and has not been already set.

Return Values

Upon successful completion, **pam_get_user** returns **PAM_SUCCESS**. Also, a reference to the user name is returned in *user*. If the routine fails, either **PAM_SYSTEM_ERR**, **PAM_BUF_ERR**, or **PAM_CONV_ERR** is returned, depending on what the actual error was, and a NULL reference in *user* is returned.

Error Codes

Item	Description
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error or failure.

pam_getenv Subroutine

Purpose

Returns the value of a defined PAM environment variable.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

char *pam_getenv (PAMHandle, VarName)
pam_handle_t *PAMHandle;
char *VarName;
```

Description

The **pam_getenv** subroutine retrieves the value of the PAM environment variable *VarName* stored in the PAM handle *PAMHandle*. Environment variables can be defined through the **pam_putenv()** call. If *VarName* is defined, its value is returned in memory allocated by the library; it is the caller's responsibility to free this memory. Otherwise, a NULL pointer is returned.

Parameters

Item	Description
PAMHandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
VarName	The name of the PAM environment variable to get the value for.

Return Values

Upon successful completion, **pam_getenv** returns the value of the *VarName* PAM environment variable. If the routine fails or *VarName* is not defined, NULL is returned.

pam_getenvlist Subroutine

Purpose

Returns a list of all of the defined PAM environment variables and their values.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

char **pam_getenvlist (PAMHandle)
pam_handle_t *PAMHandle;
```

Description

The **pam_getenvlist** subroutine returns a pointer to a list of the currently defined environment variables in the PAM handle, *PAMHandle*. Environment variables can be set through calls to the **pam_putenv()** subroutine. The library returns the environment in an allocated array in which the last entry of the array is NULL. The caller is responsible for freeing the memory of the returned list.

Parameters

Item	Description
PAMHandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.

Return Values

Upon successful completion, **pam_getenvlist** returns a pointer to a list of strings, one for each currently defined PAM environment variable. Each string is of the form *VARIABLE*=*VALUE*, where *VARIABLE* is the name of the variable and *VALUE* is its value. This list is terminated with a NULL entry. If the routine fails or there are no PAM environment variables defined, a NULL reference is returned. The caller is responsible for freeing the memory of the returned value.

pam_open_session Subroutine

Purpose

Opens a new PAM user session.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_open_session (PAMHandle, Flags)
pam_handle_t *PAMHandle;
int Flags;
```

Description

The **pam_open_session** subroutine opens a new user session for an authenticated PAM user. A call to **pam_authenticate()** is typically made prior to invoking this subroutine. Applications that open a user session should subsequently close the session with **pam_close_session()** when the session has ended.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The flags are used to set pam_acct_mgmt options. The recognized flags are: • **PAM_SILENT** – No messages should be displayed

Return Values

Upon successful completion, **pam_open_session** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_SESSION_ERR	An error occurred while creating/removing an entry for the new session.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_putenv Subroutine

Purpose

Defines a PAM environment variable.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_putenv (PAMHandle, NameValue)
pam_handle_t *PAMHandle;
const char *NameValue;
```

Description

The **pam_putenv** subroutine sets and deletes environment variables in the PAM handle, *PAMHandle*. Applications can retrieve the defined variables by calling **pam_getenv()** or **pam_getenvlist()** and add them to the user's session. If a variable with the same name is already defined, the old value is replaced by the new value.

Parameters

Item	Description
PAMHandle	The PAM authentication handle, obtained from a previous call to **pam_start()**.

Item	Description
NameValue	A string of the form *name*=*value* to be stored in the environment section of the PAM handle. The following behavior is exhibited with regards to the format of the passed-in string: ***NAME=VALUE*** Creates or overwrites the value for the variable in the environment. ***NAME=*** Sets the variable to the empty string. ***NAME*** Deletes the variable from the environment, if it is currently defined.

Return Values

Upon successful completion, **pam_putenv** returns **PAM_SUCCESS**. If the routine fails, either **PAM_SYSTEM_ERR** or **PAM_BUF_ERR** is returned, depending on the actual error.

Error Codes

Item	Description
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.

pam_set_data Subroutine

Purpose

Sets information for a specific PAM module for the active PAM session.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_set_data (PAMHandle, ModuleDataName, Data, *(cleanup)(pam_handle_t *pamh, void *data,
                  int pam_end_status))
pam_handle_t *PAMHandle;
const char *ModuleDataName;
void *Data;
void *(cleanup)(pam_handle_t *pamh, void *data, int pam_end_status);
```

Description

The **pam_set_data** subroutine allows for the setting and updating of module-specific data within the PAM handle, *PAMHandle*. The *ModuleDataName* argument serves to uniquely identify the data, *Data*. Stored information can be retrieved by specifying *ModuleDataName* and passing it, along with the appropriate PAM handle, to **pam_get_data()**. The **cleanup** argument is a pointer to a function that is called to free allocated memory used by the *Data* when **pam_end()** is invoked. If data is already associated with *ModuleDataName*, PAM does a cleanup of the old data, overwrites it with *Data*, and replaces the old **cleanup** function. If the information being set is of a known PAM item type, use the **pam_putenv** subroutine instead.

Parameters

Item	Description
PAMHandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
ModuleDataName	A unique identifier for *Data*.
Data	A reference to the data denoted by *ModuleDataName*.
cleanup	A function pointer that is called by **pam_end()** to clean up all allocated memory used by *Data*.

Return Values

Upon successful completion, **pam_set_data_** returns **PAM_SUCCESS**. If the routine fails, either **PAM_SYSTEM_ERR** or **PAM_BUF_ERR** is returned, depending on the actual error.

Error Codes

Item	Description
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.

pam_set_item Subroutine

Purpose

Sets the value of an item for this PAM session.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_set_item (PAMHandle, ItemType, Item)
pam_handle_t *PAMHandle;
int ItemType;
void **Item;
```

Description

The **pam_set_item** subroutine allows for the setting and updating of a set of known PAM items. The item value is stored within the PAM handle, *PAMHandle*. If a previous value exists for the item type, *ItemType*, then the old value is overwritten with the new value, *Item*.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.

Item	Description
ItemType	The type of item that is being requested. The following values are valid item types: • **PAM_SERVICE** – The service name requesting this PAM session. • **PAM_USER** – The user name of the user being authenticated. • **PAM_AUTHTOK** – The user's current authentication token. Interpreted as the new authentication token by password modules. • **PAM_OLDAUTHOK** – The user's old authentication token. Interpreted as the current authentication token by password modules. • **PAM_TTY** – The terminal name. • **PAM_RHOST** – The name of the remote host. • **PAM_RUSER** – The name of the remote user. • **PAM_CONV** – The **pam_conv** structure for conversing with the user. • **PAM_USER_PROMPT** – The default prompt for the user (used by **pam_get_user()**). For security, **PAM_AUTHTOK** and **PAM_OLDAUTHTOK** are only available to PAM modules.
Item	The value that the *ItemType* is set to.

Return Values

Upon successful completion, **pam_set_item** returns **PAM_SUCCESS**. If the routine fails, either **PAM_SYSTEM_ERR** or **PAM_BUF_ERR** is returned, depending on what the actual error was.

Error Codes

Item	Description
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_SYMBOL_ERR	Symbol not found.

pam_setcred Subroutine

Purpose

Establishes, changes, or removes user credentials for authentication.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_setcred (PAMHandle, Flags)
pam_handle_t *PAMHandle;
int Flags;
```

Description

The **pam_setcred** subroutine allows for the credentials of the PAM user for the current PAM session to be modified. Functions such as establishing, deleting, renewing, and refreshing credentials are defined.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The flags are used to set **pam_setcred** options. The recognized flags are: • **PAM_SILENT** – No messages should be displayed. • **PAM_ESTABLISH_CRED*** – Sets the user's credentials. This is the default. • **PAM_DELETE_CRED*** – Removes the user credentials. • **PAM_REINITIALIZE_CRED*** – Renews the user credentials. • **PAM_REFRESH_CRED*** – Refresh the user credentials, extending their lifetime. *Mutually exclusive but may be logically OR'd with **PAM_SILENT**. If one of them is not set, **PAM_ESTABLISH_CRED** is assumed.

Return Values

Upon successful completion, **pam_setcred** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_CRED_UNAVAIL	The user credentials cannot be found.
PAM_CRED_EXPIRED	The user's credentials have expired.
PAM_CRED_ERR	A failure occurred while setting user credentials.
PAM_USER_UNKNOWN	The user is not known.

Item	Description
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_sm_acct_mgmt Subroutine

Purpose

PAM module implementation for **pam_acct_mgmt()**.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_acct_mgmt (PAMHandle, Flags, Argc, Argv)
pam_handle_t *PAMHandle;
int Flags;
int Argc;
const char **Argv;
```

Description

The **pam_sm_acct_mgmt** subroutine is invoked by the PAM library in response to a call to **pam_acct_mgmt**. The **pam_sm_acct_mgmt** subroutine performs the account and password validation for a user and is associated with the "account" service in the PAM configuration file. It is up to the module writers to implement their own service-dependent version of **pam_sm_acct_mgmt**, if the module requires this feature. Actual checks performed are at the discretion of the module writer but typically include checks such as password expiration and login time validation.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The *Flags* argument can be a logically OR'd combination of the following: • **PAM_SILENT** – No messages should be displayed. • **PAM_DISALLOW_NULL_AUTHTOK** – Do not authenticate a user with a NULL authentication token.
Argc	The number of module options specified in the PAM configuration file.

Item	Description
Argv	The module options specified in the PAM configuration file. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values

Upon successful completion, **pam_sm_acct_mgmt** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_ACCT_EXPIRED	The user's account has expired.
PAM_NEW_AUTHTOKEN_REQD	The user's password needs to be changed. This is usually due to password aging or because it was last set by the system administrator. At this stage, most users can still change their passwords. Applications should call **pam_chauthtok()** and have the users change their password.
PAM_AUTHTOK_EXPIRED	The user's password has expired. Unlike **PAM_NEW_AUTHTOKEN_REQD**, the password cannot be changed by the user.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_sm_authenticate Subroutine

Purpose

PAM module-specific implementation of **pam_authenticate()**.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_authenticate (PAMHandle, Flags, Argc, Argv)
pam_handle_t *PAMHandle;
int Flags;
int Argc;
```

```
const char **Argv;
```

Description

When an application invokes **pam_authenticate()**, the PAM Framework calls **pam_sm_authenticate** for each module in the authentication module stack. This allows all the PAM module authors to implement their own authenticate routine. **pam_authenticate** and **pam_sm_authenticate** provide an authentication service to verify that the user is allowed access.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The flags are used to set **pam_acct_mgmt** options. The recognized flags are: • **PAM_SILENT** – No messages should be displayed. • **PAM_DISALLOW_NULL_AUTHTOK** – Do not authenticate a user with a NULL authentication token.
Argc	The number of module options defined.
Argv	The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values

Upon successful completion, **pam_sm_authenticate** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_AUTH_ERR	An error occurred in authentication, usually because of an invalid authentication token.
PAM_CRED_INSUFFICIENT	The user has insufficient credentials to access the authentication data.
PAM_AUTHINFO_UNAVAIL	The authentication information cannot be retrieved.
PAM_USER_UNKNOWN	The user is not known.
PAM_MAXTRIES	The maximum number of authentication retries has been reached.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.

Item	Description
PAM_PERM_DENIED	Access permission was denied to the user.

pam_sm_chauthtok Subroutine

Purpose

PAM module-specific implementation of **pam_chauthtok()**.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_chauthtok (PAMHandle, Flags, Argc, Argv)
pam_handle_t *PAMHandle;
int Flags;
int Argc;
const char **Argv;
```

Description

When an application invokes **pam_chauthtok()**, the PAM Framework calls **pam_sm_chauthtok** for each module in the password module stack. The **pam_sm_chauthtok** module interface is intended to change the user's password or authentication token. Before any password is changed, **pam_sm_chauthtok** performs preliminary tests to ensure necessary hosts and information, depending on the password service, are there. If **PAM_PRELIM_CHECK** is specified, only these preliminary checks are done. If successful, the authentication token is ready to be changed. If the **PAM_UPDATE_AUTHTOK** flag is passed in, **pam_sm_chauthtok** should take the next step and change the user's authentication token. If the **PAM_CHANGE_EXPIRED_AUTHTOK** flag is set, the module should check the authentication token for aging and expiration. If the user's authentication token is aged or expired, the module should store that information by passing it to **pam_set_data()**. Otherwise, the module should exit and return **PAM_IGNORE**. Required information is obtained through the PAM handle or by prompting the user by way of **PAM_CONV**.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.

Item	Description
Flags	The flags are used to set **pam_acct_mgmt** options. The recognized flags are: • **PAM_SILENT** – No messages should be displayed. • **PAM_CHANGE_EXPIRED_AUTHTOK** – Only expired passwords should be changed. If this flag is not included, all users using the related password service are forced to update their passwords. • **PAM_PRELIM_CHECK*** – Only perform preliminary checks to see if the password can be changed, but do not change it. • **PAM_UPDATE_AUTHTOK*** – Perform all necessary checks, and if possible, change the user's password/ authentication token. * **PAM_PRELIM_CHECK** and **PAM_UPDATE_AUTHTOK** are mutually exclusive.
Argc	The number of module options defined.
Argv	The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values

Upon successful completion, **pam_sm_chauthtok** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_AUTHTOK_ERR	A failure occurred while updating the authentication token.
PAM_TRY_AGAIN	Preliminary checks for changing the password have failed. Try again later.
PAM_AUTHTOK_RECOVERY_ERR	An error occurred while trying to recover the authentication information.
PAM_AUTHTOK_LOCK_BUSY	Cannot get the authentication token lock. Try again later
PAM_AUTHTOK_DISABLE_AGING	Authentication token aging checks are disabled and were not performed.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.

Item	Description
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_sm_close_session Subroutine

Purpose

PAM module-specific implementation to close a session previously opened by **pam_sm_open_session()**.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_close_session (PAMHandle, Flags, Argc, Argv)
pam_handle_t *PAMHandle;
int Flags;
int Argc;
const char **Argv;
```

Description

When an application invokes **pam_close_session()**, the PAM Framework calls **pam_sm_close_session** for each module in the session module stack. The **pam_sm_close_session** module interface is intended to clean up and terminate any user session started by **pam_sm_open_session()**.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The flags are used to set **pam_acct_mgmt** options. The recognized flag is: • **PAM_SILENT** – No messages should be displayed.
Argc	The number of module options defined.
Argv	The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values

Upon successful completion, **pam_sm_close_session** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_SESSION_ERR	An error occurred while creating or removing an entry for the new session.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_sm_open_session Subroutine

Purpose

PAM module-specific implementation of **pam_open_session**.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_open_session (PAMHandle, Flags, Argc, Argv)
pam_handle_t *PAMHandle;
int Flags;
int Argc;
const char **Argv;
```

Description

When an application invokes **pam_open_session()**, the PAM Framework calls **pam_sm_open_session** for each module in the session module stack. The **pam_sm_open_session** module interface starts a new user session for an authenticated PAM user. All session-specific information and memory used by opening a session should be cleaned up by **pam_sm_close_session()**.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.

Item	Description
Flags	The flags are used to set **pam_acct_mgmt** options. The recognized flag is: • **PAM_SILENT** – No messages should be displayed.
Argc	The number of module options defined.
Argv	The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values

Upon successful completion, **pam_sm_open_session** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_SESSION_ERR	An error occurred while creating or removing an entry for the new session.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_sm_setcred Subroutine

Purpose

PAM module-specific implementation of **pam_setcred**.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>
#include <security/pam_modules.h>

int pam_sm_setcred (PAMHandle, Flags, Argc, Argv)
pam_handle_t *PAMHandle;
int Flags;
int Argc;
const char **Argv;
```

Description

When an application invokes **pam_setcred()**, the PAM Framework calls **pam_sm_setcred** for each module in the authentication module stack. The **pam_sm_setcred** module interface allows for the setting of module-specific credentials in the PAM handle. The user's credentials should be set based upon the user's authentication state. This information can usually be retrieved with a call to **pam_get_data()**.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
Flags	The flags are used to set **pam_setcred** options. The recognized flags are: • **PAM_SILENT** – No messages should be displayed. • **PAM_ESTABLISH_CRED*** – Sets the user's credentials. This is the default. • **PAM_DELETE_CRED*** – Removes the user credentials. • **PAM_REINITIALIZE_CRED*** – Renews the user credentials. • **PAM_REFRESH_CRED*** – Refreshes the user credentials, extending their lifetime. *Mutually exclusive. If one of them is not set, **PAM_ESTABLISH_CRED** is assumed.
Argc	The number of module options defined.
Argv	The module options. These options are module-dependent. Any modules receiving invalid options should ignore them.

Return Values

Upon successful completion, **pam_sm_setcred** returns **PAM_SUCCESS**. If the routine fails, a different error is returned, depending on the actual error.

Error Codes

Item	Description
PAM_CRED_UNAVAIL	The user credentials cannot be found.
PAM_CRED_EXPIRED	The user's credentials have expired.
PAM_CRED_ERR	A failure occurred while setting user credentials.
PAM_USER_UNKNOWN	The user is not known.
PAM_OPEN_ERR	One of the PAM authentication modules could not be loaded.
PAM_SYMBOL_ERR	A necessary item is not available to a PAM module.
PAM_SERVICE_ERR	An error occurred in a PAM module.

Item	Description
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.
PAM_CONV_ERR	A conversation error occurred.
PAM_PERM_DENIED	Access permission was denied to the user.

pam_start Subroutine

Purpose

Initiates a new PAM user authentication session.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

int pam_start (Service, User, Conversation, PAMHandle)
const char *Service;
const char *User;
const struct pam_conv *Conversation;
pam_handle_t **PAMHandle;
```

Description

The **pam_start** subroutine begins a new PAM session for authentication within one of the four realms of the PAM environment [authentication, account, session, password]. This routine is called only at the start of the session, not at the start of each module comprising the session. The PAM handle, *PAMHandle*, returned by this subroutine is subsequently used by other PAM routines. The handle must be cleaned up at the end of use, which can easily be done by passing it as an argument to **pam_end**.

Parameters

Item	Description
Service	The name of the service initiating this PAM session.
User	The user who is being authenticated.

Item	Description
Conversation	The PAM conversation struct enabling communication with the user. This structure, **pam_conv**, consists of a pointer to a conversation function, as well as a pointer to application data.

```
struct pam_conv {
    int (**conv)();
    void (**appdata_ptr);
}
```

The argument **conv** is defined as:

```
int conv( int num_msg, const struct pam_message **msg,
          const struct pam_response **resp, void *appdata );
```

The conversation function, **conv**, allows PAM to send messages to, and get input from, a user. The arguments to the function have the following definition and behavior:

num_msg
The number of lines of messages to be displayed (all messages are returned in one-line fragments, each no longer than **PAM_MAX_MSG_SIZE** characters and with no more lines than **PAM_MAX_NUM_MSG**)

msg
Contains the message text and its style.

```
struct pam_message {
    int style;    /* Message style */
    char *msg;    /* The message */
}
```

The message style, can be one of:

PAM_PROMPT_ECHO_OFF
Prompts users with `message` and does not echo their responses; it is typically for use with requesting passwords and other sensitive information.

PAM_PROMPT_ECHO_ON
Prompts users with `message` and echoes their responses back to them.

PAM_ERROR_MSG
Displays `message` as an error message.

PAM_TEXT_INFO
Displays general information, such as authentication failures.

resp
Holds the user's response and a response code.

```
struct pam_response {
    char **resp;       /* Reference to the response */
    int resp_retcode;  /* Not used, should be 0 */
}
```

appdata, appdata_ptr
Pointers to the application data that can be passed by the calling application to the PAM modules. Use these to allow PAM to send data back to the application.

PAMHandle	The PAM handle representing the current user authentication session is returned upon successful completion.

Return Values

Upon successful completion, **pam_start** returns **PAM_SUCCESS**, and a reference to the pointer of a valid PAM handle is returned through *PAMHandle*. If the routine fails, a value different from **PAM_SUCCESS** is returned, and the *PAMHandle* reference is NULL.

Error Codes

Item	Description
PAM_SERVICE_ERR	An error occurred in a PAM module.
PAM_SYSTEM_ERR	A system error occurred.
PAM_BUF_ERR	A memory error occurred.

pam_strerror Subroutine

Purpose

Translates a PAM error code to a string message.

Library

PAM Library (**libpam.a**)

Syntax

```
#include <security/pam_appl.h>

const char *pam_strerror (PAMHandle, ErrorCode)
pam_handle_t *PAMHandle;
int ErrorCode;
```

Description

The **pam_strerror** subroutine uses the error number returned by the PAM routines and returns the PAM error message that is associated with that error number. If the error number is not known to **pam_strerror**, or there is no translation error message, then NULL is returned. The caller should not free or modify the returned string.

Parameters

Item	Description
PAMhandle	The PAM handle representing the current user authentication session. This handle is obtained by a call to **pam_start()**.
ErrorCode	The PAM error code for which the PAM error message is to be retrieved.

Return Values

Upon successful completion, **pam_strerror** returns the PAM error message corresponding to the PAM error code, *ErrorCode*. A NULL pointer is returned if the routine fails, the error code is not known, or no error message exists for that error code.

passwdexpired Subroutine

Purpose

Checks the user's password to determine if it has expired.

Syntax

```
passwdexpired ( UserName,  Message)
char *UserName;
char **Message;
```

Description

The **passwdexpired** subroutine checks a user's password to determine if it has expired. The subroutine checks the **registry** variable in the **/etc/security/user** file to ascertain where the user is administered. If the **registry** variable is not defined, the **passwdexpired** subroutine checks the local, NIS, and DCE databases for the user definition and expiration time.

The **passwdexpired** subroutine may pass back informational messages, such as how many days remain until password expiration.

Parameters

Item	Description
UserName	Specifies the user's name whose password is to be checked.
Message	Points to a pointer that the **passwdexpired** subroutine allocates memory for and fills in. This string is suitable for printing and issues messages, such as in how many days the password will expire.

Return Values

Upon successful completion, the **passwdexpired** subroutine returns a value of 0. If this subroutine fails, it returns one of the following values:

The **passwdexpired** subroutine returns 0 when the user password is set to * in the **/etc/security/passwd** file. The new *unix_passwd_compat* attribute is introduced under the **usw** stanza in the **/etc/security/login.cfg** file. When this attribute is set as true, the **passwdexpired** subroutine returns a non-zero value, compatible with other UNIX versions. The default value of this attribute is false. Valid values are true or false.

The **passwdexpired** subroutine returns a value of 2 when the user's **maxage** attribute is set to a value greater than zero and the user password is set to * in the */etc/security/passwd* file.

Item	Description
1	Indicates that the password is expired, and the user must change it.
2	Indicates that the password is expired, and only a system administrator may change it.
-1	Indicates that an internal error has occurred, such as a memory allocation (malloc) failure or database corruption.

Error Codes

The **passwdexpired** subroutine fails if one or more of the following values is true:

Item	Description
ENOENT	Indicates that the user could not be found.
EACCES	Indicates that the user did not have permission to check password expiration.
ENOMEM	Indicates that memory allocation (malloc) failed.
EINVAL	Indicates that the parameters are not valid.

passwdexpiredx Subroutine

Purpose

Checks the user's password to determine if it has expired, in multiple methods.

Syntax

```
passwdexpiredx (UserName, Message, State)
char *UserName;
char **Message;
char **State;
```

Description

The **passwdexpiredx** subroutine checks a user's password to determine if it has expired. The subroutine uses the user's **SYSTEM** attribute to ascertain which administrative domains are used for password authentication.

The **passwdexpiredx** subroutine can pass back informational messages, such as how many days remain until password expiration.

The *State* parameter can contain information about the results of the authentication process. The *State* parameter from an earlier call to the **authenticatex** subroutine can be used to control how password expiration checking is performed. Authentication mechanisms that were not used to authenticate a user are not examined for expired passwords. The *State* parameter must be initialized to reference a null pointer if the *State* parameter from an earlier call to the **authenticatex** subroutine is not used.

Parameters

Item	Description
UserName	Specifies the user's name whose password is to be checked.
Message	Points to a pointer that the **passwdexpiredx** subroutine allocates memory for and fills in. This string is suitable for printing, and it issues messages, such as an alert that indicates how many days are left before the password expires.
State	Points to a pointer that the **passwdexpiredx** subroutine allocates memory for and fills in. The *State* parameter can also be the result of an earlier call to the **authenticatex** subroutine. The *State* parameter contains information about the results of the password expiration examination process for each term in the user's **SYSTEM** attribute. The calling application is responsible for freeing this memory when it is no longer needed for a subsequent call to the **chpassx** subroutine.

Return Values

Upon successful completion, the **passwdexpiredx** subroutine returns a value of 0. If this subroutine fails, it returns one of the following values:

Item	Description
-1	Indicates that an internal error has occurred, such as a memory allocation (malloc) failure or database corruption.
1	Indicates that one or more passwords are expired, and the user must change it. None of the expired passwords require system administrator intervention to be changed.
2	Indicates that one or more passwords are expired, at least one of which must be changed by the user and at least one of which requires system administrator intervention to be changed.
3	Indicates that all expired passwords require system administrator intervention to be changed.

Error Codes

The **passwdexpiredx** subroutine fails if one or more of the following values is true:

Item	Description
EACCES	The user did not have permission to access the password attributes required to check password expiration.
EINVAL	The parameters are not valid.
ENOENT	The user could not be found.
ENOMEM	Memory allocation (malloc) failed.

passwdpolicy Subroutine

Purpose

Supports password strength policies on a per-user or per-named-policy basis.

Syntax

```
#include <pwdpolicy.h>
int passwdpolicy (const char *name, int type, const char *old_password,
                  const char *new_password, time64_t last_update);
```

Description

The **passwdpolicy** subroutine supports application use of password strength policies on a per-user or per-named-policy basis. The policies that are supported include password dictionaries, history list length, history list expiration, maximum lifetime of a password, minimum period of time between permitted password changes, maximum period after which an expired password can be changed, maximum number of repeated characters in a password, minimum number of alphabetic characters in a password, minimum number of lower case alphabetic characters in a password, minimum number of upper case alphabetic characters in a password, minimum number of digits in a password, minimum number of special characters in a password, minimum number of non-alphabetic characters in a password, minimum length of a password, and a list of loadable modules that can be used to determine additional password strength rules.

The *type* parameter allows an application to select where the policy values are located. Privileged process can use either **PWP_USERNAME** or **PWP_SYSTEMPOLICY**. Unprivileged processes are limited to using **PWP_LOCALPOLICY**.

The following named attributes are used for each test:

Item	Description
dictionlist	A **SEC_LIST** value that gives a list of dictionaries to be checked. If *new_password* is found in any of the named dictionaries, this test fails. If this test fails, the return value contains the **PWP_IN_DICTIONARY** bit.
histsize	A **SEC_INT** value giving the permissible size of the named user's password history. The named user's password history is obtained by calling **getuserattr** with the **S_HISTLIST** attribute. If this attribute does not exist, password history checks are disabled. A value of 0 disables password history tests. If this test fails, the return value contains the **PWP_REUSED_PW** bit.
histexpire	A **SEC_INT** value that gives the number of weeks that must elapse before a password in the named user's password history list can be reused. If this test fails the return value contains the **PWP_REUSED_TOO_SOON** bit.
maxage	A **SEC_INT** value that gives the number of weeks a password can be considered valid. A password that has not been modified more recently than **maxage** weeks is considered to have expired and is subject to the **maxexpired** test. A value less than or equal to 0 disables this test. This attribute is used to determine if **maxexpired** must be tested, and it does not generate a return value.
minage	A **SEC_INT** value that gives the number of weeks before a password can be changed. A password that has been modified more recently than **minage** weeks fails this test. A value less than or equal to 0 disables this test. If this test fails, the return value contains the **PWP_TOO_SOON** bit.
maxexpired	A **SEC_INT** value that gives the number of weeks after which an expired password cannot be changed. A value of 0 indicates that an expired password cannot be changed. A value of -1 indicates that an expired password can be changed after any length of time. If this test fails, the return value contains the **PWP_EXPIRED** bit.
maxrepeats	A **SEC_INT** value that gives the maximum number of times any single character can appear in the new password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the **PWP_TOO_MANY_REPEATS** bit.
mindiff	A **SEC_INT** value that gives the maximum number of characters in the new password that must not be present in the old password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the **PWP_TOO_MANY_SAME** bit.
minalpha	A **SEC_INT** value that gives the minimum number of alphabetic characters that must be present in the password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the **PWP_TOO_FEW_ALPHA** bit.
minother	A **SEC_INT** value that gives the minimum number of nonalphabetic characters that must be present in the password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the bit **PWP_TOO_FEW_OTHER** bit.
minlen	A **SEC_INT** value that gives the minimum required length of a password. There is no maximum value for this attribute. A value less than or equal to 0 disables this test. If this test fails, the return value contains the **PWP_TOO_SHORT** bit.
pwdchecks	A **SEC_LIST** value that gives a list of named loadable modules that must be executed to validate the password. If this test fails, the return value contains the **PWP_FAILED_OTHER** bit.

Item	Description
minloweralpha	A **SEC_INT** value that gives the minimum number of lower case alphabetic characters that must be present in the password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the **PWP_TOO_FEW_LOWERALPHA** bit.
minupperalpha	A **SEC_INT** value that gives the minimum number of upper case alphabetic characters that must be present in the password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the **PWP_TOO_FEW_UPPERALPHA** bit.
mindigit	A **SEC_INT** value that gives the minimum number of digits that must be present in the password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the **PWP_TOO_FEW_DIGIT** bit.
minspecialchar	A **SEC_INT** value that gives the minimum number of special characters that must be present in the password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the**PWP_TOO_FEW_SPECIALCHAR** bit.

Parameters

Item	Description
name	The name of either a specific user or named policy. User names have policy information determined by invoking the **getuserattr** subroutine. Policy names have policy information determined by invoking the **getconfattr** subroutine.
type	One of three values: **PWP_USERNAME** Policy values for **PWP_USERNAME** are stored in **/etc/security/user**. Password tests **PWP_REUSED_PW** and **PWP_REUSED_TOO_SOON** are only enabled for this value. **PWP_SYSTEMPOLICY** Policy values for **PWP_SYSTEMPOLICY** are stored in **/etc/security/passwd_policy**. **PWP_LOCALPOLICY** Policy values for **PWP_LOCALPOLICY** are stored in **/usr/lib/security/passwd_policy**.
old_password	The current value of the password. This function does not verify that *old_password* is the correct current password. Invoking **passwdpolicy** with a NULL pointer for this parameter disables **PWP_TOO_MANY_SAME** tests.
new_password	The value of the new password. Invoking **passwdpolicy** with a NULL pointer for this parameter disables all tests except password age tests.
last_update	The time the password was last changed, as a **time64_t** value, expressed in seconds since the UNIX epoch. A 0 value for this parameter disables password age tests regardless of the value of any other parameter.

Return Values

The return value is a bit map representation of the tests that failed. A return value of 0 indicates that all password rules passed. A value of -1 indicates that some other error, other than a failed password test, has occurred. The **errno** variable indicates the cause of that error. Applications must compare a nonzero return value against -1 before checking any specific bits in the return value.

Files

The **/usr/include/pwdpolicy.h** header file.

passwdstrength Subroutine

Purpose

Performs basic password age and construction tests.

Syntax

```
#include <pwdpolicy.h>
int passwdstrength (const char *old_password, const char *new_password,
                    time64_t last_update, passwd_policy_t *policy, int checks);
```

Description

The **passwdstrength** subroutine performs basic password age and construction tests. Password history, reuse, and dictionary tests are not performed. The values contained in the *policy* parameter are used to validate the value of *new_password*.

The following fields are used by the **passwdstrength** subroutine.

Item	Description
pwp_version	Specifies the version of the **passwd_policy_t** structure. The current structure version number is **PWP_VERSION_1**.
pwp_minage	The number of seconds, as a **time32_t**, between the time a password is modified and the time the password can again be modified. This field is referenced if **PWP_TOO_SOON** is set in *checks*.
pwp_maxage	The number of seconds, as a **time32_t**, after which a password that has been modified is considered to be expired. This field is referenced if **PWP_EXPIRED** is set in *checks*.
pwp_maxexpired	The number of seconds, as a **time32_t**, since a password has expired after which it can no longer be modified. A value of 0 indicates that an expired password cannot be changed. A value of -1 indicates that an expired password can be changed after any length of time. This field is referenced if **PWP_EXPIRED** is set in *checks*.
pwp_minalpha	The minimum number of characters in the password that must be alphabetic characters, as determined by invoking the **isalpha()** macro. A value less than or equal to 0 disables this test. This field is referenced if **PWP_TOO_FEW_ALPHA** is set in *checks*.
pwp_minother	The minimum number of characters in the password that cannot be alphabetic characters, as determined by invoking the **isalpha()** macro. A value less than or equal to 0 disables this test. This field is referenced if **PWP_TOO_FEW_OTHER** is set in *checks*.
pwp_minlen	The minimum total number of characters in the password. A value less than or equal to 0 disables this test. This field is referenced if **PWP_TOO_SHORT** is set in *checks*.
pwp_maxrepeats	The maximum number of times an individual character can appear in the password. A value less than or equal to 0 disables this test. This field is referenced if **PWP_TOO_MANY_REPEATS** is set in *checks*.

Item	Description
pwp_mindiff	The minimum number of characters that must be changed between the old password and the new password. A value less than or equal to 0 disables this test. If this test fails, the return value contains the bit **PWP_TOO_MANY_SAME**. This field is referenced if **PWP_TOO_MANY_SAME** is set in *checks*.

Parameters

Item	Description
old_password	The value of the current password. This parameter must be non-NULL if **PWP_TOO_MANY_SAME** is set in *checks* or the results are undefined.
new_password	The value of the new password. This parameter must be non-NULL if any of **PWP_TOO_SHORT**, **PWP_TOO_FEW_ALPHA**, **PWP_TOO_FEW_OTHER**, **PWP_TOO_MANY_SAME**, or **PWP_TOO_MANY_REPEATS** are set in *checks* or the results are undefined.
last_update	The time the password was last changed, as a **time64_t** value, expressed in seconds since the UNIX epoch. A 0 value for this parameter indicates that the password has never been set. This might cause **PWP_EXPIRED** to be set in the return value if it is set in *checks*.
policy	A pointer to a **passwd_policy_t** containing the values for the password policy attributes.
checks	A bitmask value that indicates the set of password tests to be performed. The return value contains only those bits that are defined in *checks*.

Return Values

The return value is a bit-mapped representation of the tests that failed. A return value of 0 indicates that all password rules requested in the *checks* parameter passed. A value of -1 indicates that some other error, other than a password test, has occurred. The **errno** variable indicates the cause of that error. Applications must compare a non-zero return value against -1 before checking any specific bits in the return value.

Files

The **/usr/include/pwdpolicy.h** header file.

pathconf or fpathconf Subroutine

Purpose

Retrieves file-implementation characteristics.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
long pathconf ( Path,  Name)
const char *Path;
int Name;

long fpathconf( FileDescriptor, Name)
int FileDescriptor, Name;
```

Description

The **pathconf** subroutine allows an application to determine the characteristics of operations supported by the file system contained by the file named by the *Path* parameter. Read, write, or execute permission of the named file is not required, but all directories in the path leading to the file must be searchable.

The **fpathconf** subroutine allows an application to retrieve the same information for an open file.

Parameters

Item	Description
Path	Specifies the path name.
FileDescriptor	Specifies an open file descriptor.

Item	Description
Name	Specifies the configuration attribute to be queried. If this attribute is not applicable to the file specified by the *Path* or *FileDescriptor* parameter, the **pathconf** subroutine returns an error. Symbolic values for the *Name* parameter are defined in the **unistd.h** file: **_PC_LINK_MAX** Specifies the maximum number of links to the file. **_PC_MAX_CANON** Specifies the maximum number of bytes in a canonical input line. This value is applicable only to terminal devices. **_PC_MAX_INPUT** Specifies the maximum number of bytes allowed in an input queue. This value is applicable only to terminal devices. **_PC_NAME_MAX** Specifies the maximum number of bytes in a file name, not including a terminating null character. This number can range from 14 through 255. This value is applicable only to a directory file. **_PC_PATH_MAX** Specifies the maximum number of bytes in a path name, including a terminating null character. **_PC_PIPE_BUF** Specifies the maximum number of bytes guaranteed to be written atomically. This value is applicable only to a first-in-first-out (FIFO). **_PC_CHOWN_RESTRICTED** Returns 0 if the use of the **chown** subroutine is restricted to a process with appropriate privileges, and if the **chown** subroutine is restricted to changing the group ID of a file only to the effective group ID of the process or to one of its supplementary group IDs. If XPG_SUS_ENV is set to ON, the **_PC_CHOWN_RESTRICTED** returns a value greater than zero. **_PC_NO_TRUNC** Returns 0 if long component names are truncated. This value is applicable only to a directory file. If XPG_SUS_ENV is set to ON, the **_PC_NO_TRUNC** returns a value greater than zero. **_PC_VDISABLE** This is always 0. No disabling character is defined. This value is applicable only to a terminal device. **_PC_AIX_DISK_PARTITION** Determines the physical partition size of the disk. **Note:** The **_PC_AIX_DISK_PARTITION** variable is available only to the root user. **_PC_AIX_DISK_SIZE** Determines the disk size in megabytes. **Note:** The **_PC_AIX_DISK_SIZE** variable is available only to the root user. **_PC_FILESIZEBITS** Returns the minimum number of bits required to hold the file system's maximum file size as a signed integer. The smallest value returned is **32**. **_PC_SYNC_IO** Returns **-1** if the file system does not support the **Synchronized Input and Output** option. Any value other than **-1** is returned if the file system supports the option.

Note:

1. If the *Name* parameter has a value of **_PC_LINK_MAX**, and if the *Path* or *FileDescriptor* parameter refers to a directory, the value returned applies to the directory itself.
2. If the *Name* parameter has a value of **_PC_NAME_MAX** or **_PC_NO_TRUNC**, and if the *Path* or *FileDescriptor* parameter refers to a directory, the value returned applies to filenames within the directory.
3. If the *Name* parameter has a value if **_PC_PATH_MAX**, and if the *Path* or *FileDescriptor* parameter refers to a directory that is the working directory, the value returned is the maximum length of a relative pathname.
4. If the *Name* parameter has a value of **_PC_PIPE_BU**F, and if the *Path* parameter refers to a FIFO special file or the *FileDescriptor* parameter refers to a pipe or a FIFO special file, the value returned applies to the referenced object. If the *Path* or *FileDescriptor* parameter refers to a directory, the value returned applies to any FIFO special file that exists or can be created within the directory.
5. If the *Name* parameter has a value of **_PC_CHOWN_RESTRICTED**, and if the *Path* or *FileDescriptor* parameter refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.

Return Values

If the **pathconf** or **fpathconf** subroutine is successful, the specified parameter is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error. If the variable corresponding to the *Name* parameter has no limit for the *Path* parameter or the *FileDescriptor* parameter, both the **pathconf** and **fpathconf** subroutines return a value of -1 without changing the **errno** global variable.

Error Codes

The **pathconf** or **fpathconf** subroutine fails if the following error occurs:

Item	Description
EINVAL	The name parameter specifies an unknown or inapplicable characteristic.

The **pathconf** subroutine can also fail if any of the following errors occur:

Item	Description
EACCES	Search permission is denied for a component of the path prefix.
EINVAL	The implementation does not support an association of the *Name* parameter with the specified file.
ENAMETOOLONG	The length of the *Path* parameter string exceeds the **PATH_MAX** value.
ENAMETOOLONG	*Pathname* resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.
ENOENT	The named file does not exist or the *Path* parameter points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.
ELOOP	Too many symbolic links were encountered in resolving path.

The **fpathconf** subroutine can fail if either of the following errors occur:

Item	Description
EBADF	The *File Descriptor* parameter is not valid.
EINVAL	The implementation does not support an association of the *Name* parameter with the specified file.

pause Subroutine

Purpose

Suspends a process until a signal is received.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
int pause (void)
```

Description

The **pause** subroutine suspends the calling process until it receives a signal. The signal must not be one that is ignored by the calling process. The **pause** subroutine does not affect the action taken upon the receipt of a signal.

Return Values

If the signal received causes the calling process to end, the **pause** subroutine does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function, the calling process resumes execution from the point of suspension. The **pause** subroutine returns a value of -1 and sets the **errno** global variable to **EINTR**.

pcap_close Subroutine

Purpose

Closes the open files related to the packet capture descriptor and frees the memory used by the packet capture descriptor.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

void pcap_close(pcap_t * p);
```

Description

The **pcap_close** subroutine closes the files associated with the packet capture descriptor and deallocates resources. If the **pcap_open_offline** subroutine was previously called, the **pcap_close** subroutine closes the *savefile*, a previously saved packet capture data file. Or the **pcap_close** subroutine closes the packet capture device if the **pcap_open_live** subroutine was previously called.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned by the **pcap_open_live** or the **pcap_open_offline** subroutine.

pcap_compile Subroutine

Purpose

Compiles a filter expression into a filter program.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_compile(pcap_t * p, struct bpf_ program *fp, char * str,
int  optimize, bpf_u_int32  netmask);
```

Description

The **pcap_compile** subroutine is used to compile the string *str* into a filter program. This filter program will then be used to filter, or select, the desired packets.

Parameters

Item	Description
netmask	Specifies the *netmask* of the network device. The *netmask* can be obtained from the **pcap_lookupnet** subroutine.
optimize	Controls whether optimization on the resulting code is performed.
p	Points to a packet capture descriptor returned from the **pcap_open_offline** or the **pcap_open_live** subroutine.
program	Points to a **bpf_program** struct which will be filled in by the **pcap_compile** subroutine if the subroutine is successful.
str	Contains the filter expression.

Return Values

Upon successful completion, the **pcap_compile** subroutine returns 0, and the program parameter will hold the filter program. If **pcap_compile** subroutine is unsuccessful, -1 is returned.

pcap_datalink Subroutine

Purpose

Obtains the link layer type (data link type) for the packet capture device.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_datalink(pcap_t * p);
```

Description

The **pcap_datalink** subroutine returns the link layer type of the packet capture device, for example, IFT_ETHER. This is useful in determining the size of the datalink header at the beginning of each packet that is read.

Parameters

Item	Description
p	Points to the packet capture descriptor as returned by the **pcap_open_live** or the **pcap_open_offline** subroutine.

Return Values

The **pcap_datalink** subroutine returns the values of standard **libpcap** link layer type from the <**net/bpf.h**> header file.

Note: Only call this subroutine after successful calls to either the **pcap_open_live** or the **pcap_open_offline** subroutine. Never call the **pcap_datalink** subroutine after a call to **pcap_close** as unpredictable results will occur.

pcap_dispatch Subroutine

Purpose

Collects and processes packets.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_dispatch(pcap_t * p, int  cnt, pcap_handler  callback,
   u_char * user);
```

Description

The **pcap_dispatch** subroutine reads and processes packets. This subroutine can be called to read and process packets that are stored in a previously saved packet capture data file, known as the *savefile*. The subroutine can also read and process packets that are being captured live.

Notice that the third parameter, *callback*, is of the type **pcap_handler**. This is a pointer to a user-provided subroutine with three parameters. Define this user-provided subroutine as follows:

```
void user_routine(u_char *user, struct pcap_pkthdr *phdr, u_char *pdata)
```

The parameter, *user*, is the *user* parameter that is passed into the **pcap_dispatch** subroutine. The parameter, *phdr*, is a pointer to the **pcap_pkthdr** structure which precedes each packet in the *savefile*. The parameter, *pdata*, points to the packet data. This allows users to define their own handling of packet capture data.

Parameters

Item	Description

callback	Points to a user-provided routine that will be called for each packet read. The user is responsible for providing a valid pointer, and that unpredictable results can occur if an invalid pointer is supplied.
	
	Note: The **pcap_dump** subroutine can also be specified as the *callback* parameter. If this is done, the **pcap_dump_open** subroutine should be called first. The pointer to the **pcap_dumper_t** struct returned from the **pcap_dump_open** subroutine should be used as the *user* parameter to the **pcap_dispatch** subroutine. The following program fragment illustrates this use:
	
	`pcap_dumper_t *pd`
	`pcap_t * p;`
	`int rc = 0;`
	
	`pd = pcap_dump_open(p, "/tmp/savefile");`
	
	`rc = pcap_dispatch(p, 0 , pcap_dump, (u_char *) pd);`

cnt	Specifies the maximum number of packets to process before returning. A *cnt* of -1 processes all the packets received in one buffer. A *cnt* of 0 processes all packets until an error occurs, EOF is reached, or the read times out (when doing live reads and a non-zero read timeout is specified).
p	Points to a packet capture descriptor returned from the **pcap_open_offline** or the **pcap_open_live** subroutine. This will be used to store packet data that is read in.
user	Specifies the first argument to pass into the *callback* routine.

Return Values

Upon successful completion, the **pcap_dispatch** subroutine returns the number of packets read. If EOF is reached in a *savefile*, zero is returned. If the **pcap_dispatch** subroutine is unsuccessful, -1 is returned. In this case, the **pcap_geterr** or **pcap_perror** subroutine can be used to get the error text.

pcap_dump Subroutine

Purpose

Writes packet capture data to a binary file.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

void pcap_dump(u_char * user, struct pcap_pkthdr * h, u_char * sp);
```

Description

The **pcap_dump** subroutine writes the packet capture data to a binary file. The packet header data, contained in *h*, will be written to the the file pointed to by the *user* file pointer, followed by the packet data from *sp*. Up to `h->caplen` bytes of *sp* will be written.

The file that *user* points to (where the **pcap_dump** subroutine writes to) must be open. To open the file and retrieve its pointer, use the **pcap_dump_open** subroutine.

The calling arguments for the **pcap_dump** subroutine are suitable for use with **pcap_dispatch** subroutine and the **pcap_loop** subroutine. To retrieve this data, the **pcap_open_offline** subroutine can be invoked with the name of the file that *user* points to as its first parameter.

Parameters

Item	Description
h	Contains the packet header data that will be written to the packet capture date file, known as the *savefile*. This data will be written ahead of the rest of the packet data.
sp	Points to the packet data that is to be written to the *savefile*.
user	Specifies the *savefile* file pointer which is returned from the **pcap_dump_open** subroutine. It should be cast to a u_char * when passed in.

pcap_dump_close Subroutine

Purpose

Closes a packet capture data file, know as a *savefile*.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

void pcap_dump_close(pcap_dumper_t * p);
```

Description

The **pcap_dump_close** subroutine closes a packet capture data file, known as the *savefile*, that was opened using the **pcap_dump_open** subroutine.

Parameters

Item	Description
p	Points to a **pcap_dumper_t**, which is synonymous with a FILE *, the file pointer of a *savefile*.

pcap_dump_open Subroutine

Purpose

Opens or creates a file for writing packet capture data.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

pcap_dumper_t *pcap_dump_open(pcap_t * p, char * fname);
```

Description

The **pcap_dump_open** subroutine opens or creates the packet capture data file, known as the *savefile*. This action is specified through the *fname* parameter. The subroutine then writes the required packet capture file header to the file. The **pcap_dump** subroutine can then be called to write the packet capture data associated with the packet capture descriptor, **p**, into this file. The **pcap_dump_open** subroutine must be called before calling the **pcap_dump** subroutine.

Parameters

Item	Description
fname	Specifies the name of the file to open. A "-" indicates that standard output should be used instead of a file.
p	Specifies a packet capture descriptor returned by the **pcap_open_offline** or the **pcap_open_live** subroutine.

Return Values

Upon successful completion, the **pcap_dump_open** subroutine returns a pointer to a the file that was opened or created. This pointer is a pointer to a **pcap_dumper_t**, which is synonymous with FILE *. See the **pcap_dump** , **pcap_dispatch**, or the **pcap_loop** subroutine for an example of how to use **pcap_dumper_t**. If the **pcap_dump_open** subroutine is unsuccessful, Null is returned. Use the **pcap_geterr** subroutine to obtain the specific error text.

pcap_file Subroutine

Purpose

Obtains the file pointer to the *savefile*, a previously saved packed capture data file.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

FILE *pcap_file(pcap_t * p);
```

Description

The **pcap_file** subroutine returns the file pointer to the *savefile*. If there is no open *savefile*, 0 is returned. This subroutine should be called after a successful call to the **pcap_open_offline** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned by the **pcap_open_offline** subroutine.

Return Values

The **pcap_file** subroutine returns the file pointer to the *savefile*.

pcap_fileno Subroutine

Purpose

Obtains the descriptor for the packet capture device.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_fileno(pcap_t * p);
```

Description

The **pcap_fileno** subroutine returns the descriptor for the packet capture device. This subroutine should be called only after a successful call to the **pcap_open_live** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned by the **pcap_open_live** subroutine.

Return Values

The **pcap_fileno** subroutine returns the descriptor for the packet capture device.

pcap_geterr Subroutine

Purpose

Obtains the most recent pcap error message.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

char *pcap_geterr(pcap_t * p);
```

Description

The **pcap_geterr** subroutine returns the error text pertaining to the last pcap library error. This subroutine is useful in obtaining error text from those subroutines that do not return an error string. Since the pointer returned points to a memory space that will be reused by the pcap library subroutines, it is important to copy this message into a new buffer if the error text needs to be saved.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned by the **pcap_open_live** or the **pcap_open_offline** subroutine.

Return Values

The **pcap_geterr** subroutine returns a pointer to the most recent error message from a pcap library subroutine. If there were no previous error messages, a string with 0 as the first byte is returned.

pcap_is_swapped Subroutine

Purpose

Reports if the byte order of the previously saved packet capture data file, known as the *savefile* was swapped.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_is_swapped(pcap_t * p);
```

Description

The **pcap_is_swapped** subroutine returns 1 (True) if the current *savefile* uses a different byte order than the current system. This subroutine should be called after a successful call to the **pcap_open_offline** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned from the **pcap_open_offline** subroutine.

Return Values

Item	Description
1	If the byte order of the *savefile* is different from that of the current system.
0	If the byte order of the *savefile* is the same as that of the current system.

pcap_lookupdev Subroutine

Purpose

Obtains the name of a network device on the system.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

char *pcap_lookupdev(char * errbuf);
```

Description

The **pcap_lookupdev** subroutine gets a network device suitable for use with the **pcap_open_live** and the **pcap_lookupnet** subroutines. If no interface can be found, or none are configured to be up, Null is returned. In the case of multiple network devices attached to the system, the **pcap_lookupdev** subroutine returns the first one it finds to be up, other than the loopback interface. (Loopback is always ignored.)

Parameters

Item	Description
errbuf	Returns error text and is only set when the **pcap_lookupdev** subroutine fails.

Return Values

Upon successful completion, the **pcap_lookupdev** subroutine returns a pointer to the name of a network device attached to the system. If **pcap_lookupdev** subroutine is unsuccessful, Null is returned, and text indicating the specific error is written to *errbuf*.

pcap_lookupnet Subroutine

Purpose

Returns the network address and subnet mask for a network device.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_lookupnet(char * device, bpf_u_int32 * netp, bpf_u_int32 * maskp,
char * errbuf);
```

Description

Use the **pcap_lookupnet** subroutine to determine the network address and subnet mask for the network device, **device**.

Parameters

Item	Description
device	Specifies the name of the network device to use for the network lookup, for example, en0.
errbuf	Returns error text and is only set when the **pcap_lookupnet** subroutine fails.
maskp	Holds the subnet mask associated with **device**.
netp	Holds the network address for the **device**.

Return Values

Upon successful completion, the **pcap_lookupnet** subroutine returns 0. If the **pcap_lookupnet** subroutine is unsuccessful, -1 is returned, and *errbuf* is filled in with an appropriate error message.

pcap_loop Subroutine

Purpose

Collects and processes packets.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_loop(pcap_t * p, int  cnt, pcap_handler  callback,
   u_char * user);
```

Description

The **pcap_loop** subroutine reads and processes packets. This subroutine can be called to read and process packets that are stored in a previously saved packet capture data file, known as the *savefile*. The subroutine can also read and process packets that are being captured live.

This subroutine is similar to **pcap_dispatch** subroutine except it continues to read packets until *cnt* packets have been processed, EOF is reached (in the case of offline reading), or an error occurs. It does not return when live read timeouts occur. That is, specifying a non-zero read timeout to the **pcap_open_live** subroutine and then calling the **pcap_loop** subroutine allows the reception and processing of any packets that arrive when the timeout occurs.

Notice that the third parameter, *callback*, is of the type **pcap_handler**. This is a pointer to a user-provided subroutine with three parameters. Define this user-provided subroutine as follows:

```
void user_routine(u_char *user, struct pcap_pkthdr *phrd, u_char *pdata)
```

The parameter, *user*, will be the user parameter that was passed into the **pcap_dispatch** subroutine. The parameter, *phdr*, is a pointer to the **pcap_pkthdr** structure, which precedes each packet in the *savefile.* The parameter, *pdata*, points to the packet data. This allows users to define their own handling of their filtered packets.

Parameters

Item	Description

callback	Points to a user-provided routine that will be called for each packet read. The user is responsible for providing a valid pointer, and that unpredictable results can occur if an invalid pointer is supplied.
	
	Note: The **pcap_dump** subroutine can also be specified as the callback parameter. If this is done, call the **pcap_dump_open** subroutine first. Then use the pointer to the **pcap_dumper_t** struct returned from the **pcap_dump_open** subroutine as the user parameter to the **pcap_dispatch** subroutine. The following program fragment illustrates this use:
	
	`pcap_dumper_t *pd`
	`pcap_t * p;`
	`int rc = 0;`
	
	`pd = pcap_dump_open(p, "/tmp/savefile");`
	
	`rc = pcap_dispatch(p, 0 , pcap_dump, (u_char *) pd);`

cnt	Specifies the maximum number of packets to process before returning. A negative value causes the **pcap_loop** subroutine to loop forever, or until EOF is reached or an error occurs. A *cnt* of 0 processes all packets until an error occurs or EOF is reached.
p	Points to a packet capture descriptor returned from the **pcap_open_offline** or the **pcap_open_live** subroutine. This will be used to store packet data that is read in.
user	Specifies the first argument to pass into the *callback* routine.

Return Values

Upon successful completion, the **pcap_loop** subroutine returns 0. 0 is also returned if EOF has been reached in a *savefile*. If the **pcap_loop** subroutine is unsuccessful, -1 is returned. In this case, the **pcap_geterr** subroutine or the **pcap_perror** subroutine can be used to get the error text.

pcap_major_version Subroutine

Purpose

Obtains the major version number of the packet capture format used to write the *savefile*, a previously saved packet capture data file.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_major_version(pcap_t * p);
```

Description

The **pcap_major_version** subroutine returns the major version number of the packet capture format used to write the *savefile*. If there is no open *savefile*, 0 is returned.

Note: This subroutine should be called only after a successful call to **pcap_open_offline** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned from **pcap_open_offline** subroutine.

Return Values

The major version number of the packet capture format used to write the *savefile*.

pcap_minor_version Subroutine

Purpose

Obtains the minor version number of the packet capture format used to write the *savefile*.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_minor_version(pcap_t * p);
```

Description

The **pcap_minor_version** subroutine returns the minor version number of the packet capture format used to write the *savefile*. This subroutine should only be called after a successful call to the **pcap_open_offline** subroutine and before any calls to the **pcap_close** subroutine.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned from the **pcap_open_offline** subroutine.

Return Values

The minor version number of the packet capture format used to write the *savefile*.

pcap_next Subroutine

Purpose

Obtains the next packet from the packet capture device.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

u_char *pcap_next(pcap_t * p, struct pcap_pkthdr * h);
```

Description

The **pcap_next** subroutine returns a u_char pointer to the next packet from the packet capture device. The packet capture device can be a network device or a *savefile* that contains packet capture data. The data has the same format as used by **tcpdump**.

Parameters

Item	Description
h	Points to the packet header of the packet that is returned. This is filled in upon return by this routine.
p	Points to the packet capture descriptor to use as returned by the **pcap_open_live** or the **pcap_open_offline** subroutine.

Return Values

Upon successful completion, the **pcap_next** subroutine returns a pointer to a buffer containing the next packet and fills in the *h*, which points to the packet header of the returned packet. If the **pcap_next** subroutine is unsuccessful, Null is returned.

pcap_open_live Subroutine

Purpose

Opens a network device for packet capture.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

pcap_t *pcap_open_live( const char * device, const int snaplen,
const int promisc, const int to_ms, char * ebuf);
```

Description

The **pcap_open_live** subroutine opens the specified network device for packet capture. The term "live" is to indicate that a network device is being opened, as opposed to a file that contains packet capture data. This subroutine must be called before any packet capturing can occur. All other routines dealing with packet capture require the packet capture descriptor that is created and initialized with this routine. See the **pcap_open_offline** subroutine for more details on opening a previously saved file that contains packet capture data.

Parameters

Item	Description
device	Specifies a string that contains the name of the network device to open for packet capture, for example, en0.
ebuf	Returns error text and is only set when the **pcap_open_live** subroutine fails.
promisc	Specifies that the device is to be put into promiscuous mode. A value of 1 (True) turns promiscuous mode on. If this parameter is 0 (False), the device will remain unchanged. In this case, if it has already been set to promiscuous mode (for some other reason), it will remain in this mode.
snaplen	Specifies the maximum number of bytes to capture per packet.
to_ms	Specifies the read timeout in milliseconds.

Return Values

Upon successful completion, the **pcap_open_live** subroutine will return a pointer to the packet capture descriptor that was created. If the **pcap_open_live** subroutine is unsuccessful, Null is returned, and text indicating the specific error is written into the *ebuf* buffer.

pcap_open_live_sb Subroutine

Purpose

Opens a network device for packet capture, allowing you to specify the buffer length of a Berkeley Packet Filter (BPF).

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>
pcap_t * pcap_open_live_sb( const char *device, int snaplen,
int promisc, int to_ms, char *ebuf, int buflen )
```

Description

The **pcap_open_live_sb** subroutine opens the specified network device for packet capture. This subroutine allows you to specify the buffer size for the BPF to use in capturing the packets. You must run this subroutine before any packet capturing can occur. All other subroutines dealing with packet capture require the packet capture descriptor that is created and initialized with this subroutine.

To opening a previously saved file that contains packet capture data, use the **pcap_open_offline** subroutine.

Parameters

Item	Description
buf_len	Specifies the buffer size that the BPF is to use. If the system cannot provide memory of this size, the system will choose a smaller size.
device	Specifies a string that contains the name of the network device to open for packet capture, for example, en0.
ebuf	Returns error text and is only set when the **pcap_open_live** subroutine fails.
promisc	Specifies that the device is to be put into the promiscuous mode. A value of 1 (True) turns the promiscuous mode on. If this parameter is zero (False), the device remains unchanged. In this case, if it has already been set to the promiscuous mode (for some other reason), it remains in this mode.
snaplen	Specifies the maximum number of bytes to capture per packet.
to_ms	Specifies the read timeout in milliseconds.

Return Values

If successful, the **pcap_open_live_sb** subroutine returns a pointer to the packet capture descriptor that is created. If the **pcap_open_live_sb** subroutine is unsuccessful, NULL is returned, and the text indicating the specific error is written into the *ebuf* buffer.

pcap_open_offline Subroutine

Purpose

Opens a previously saved file containing packet capture data.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

pcap_t *pcap_open_offline(char * fname, char * ebuf);
```

Description

The **pcap_open_offline** subroutine opens a previously saved packet capture data file, known as the *savefile*. This subroutine creates and initializes a packet capture (pcap) descriptor and opens the specified *savefile* containing the packet capture data for reading.

This subroutine should be called before any other related routines that require a packet capture descriptor for offline packet processing. See the **pcap_open_live** subroutine for more details on live packet capture.

Note: The format of the *savefile* is expected to be the same as the format used by the **tcpdump** command.

Parameters

Item	Description
ebuf	Returns error text and is only set when the **pcap_open_offline** subroutine fails.
fname	Specifies the name of the file to open. A hyphen (-) passed as the *fname* parameter indicates that stdin should be used as the file to open.

Return Values

Upon successful completion, the **pcap_open_offline** subroutine will return a pointer to the newly created packet capture descriptor. If the **pcap_open_offline** subroutine is unsuccessful, Null is returned, and text indicating the specific error is written into the *ebuf* buffer.

pcap_perror Subroutine

Purpose

Prints the passed-in prefix, followed by the most recent error text.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

void pcap_perror(pcap_t * p, char * prefix);
```

Description

The **pcap_perror** subroutine prints the text of the last pcap library error to stderr, prefixed by *prefix*. If there were no previous errors, only *prefix* is printed.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned by the **pcap_open_live** subroutine or the **pcap_open_offline** subroutine.
prefix	Specifies the string that is to be printed before the stored error message.

pcap_setfilter Subroutine

Purpose

Loads a filter program into a packet capture device.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_setfilter(pcap_t * p, struct bpf_program * fp);
```

Description

The **pcap_setfilter** subroutine is used to load a filter program into the packet capture device. This causes the capture of the packets defined by the filter to begin.

Parameters

Item	Description
fp	Points to a filter program as returned from the **pcap_compile** subroutine.
p	Points to a packet capture descriptor returned from the **pcap_open_offline** or the **pcap_open_live** subroutine.

Return Values

Upon successful completion, the **pcap_setfilter** subroutine returns 0. If the **pcap_setfilter** subroutine is unsuccessful, -1 is returned. In this case, the **pcap_geterr** subroutine can be used to get the error text, and the **pcap_perror** subroutine can be used to display the text.

pcap_snapshot Subroutine

Purpose

Obtains the number of bytes that will be saved for each packet captured.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

int pcap_snapshot( pcap_t * p);
```

Description

The **pcap_snapshot** subroutine returns the snapshot length, which is the number of bytes to save for each packet captured.

Note: This subroutine should only be called after successful calls to either the **pcap_open_live** subroutine or **pcap_open_offline** subroutine. It should not be called after a call to the **pcap_close** subroutine.

Parameters

Item	Description
p	Points to the packet capture descriptor as returned by the **pcap_open_live** or the **pcap_open_offline** subroutine.

Return Values

The **pcap_snapshot** subroutine returns the snapshot length.

pcap_stats Subroutine

Purpose

Obtains packet capture statistics.

Library

pcap Library (**libpcap.a**)

Syntax

#include<**pcap.h**> **int pcap_stats (pcap_t ***p***, struct pcap_stat ***ps***)**;

Description

The **pcap_stats** subroutine fills in a **pcap_stat** struct. The values represent packet statistics from the start of the run to the time of the call. Statistics for both packets that are received by the filter and packets that are dropped are stored inside a **pcap_stat** struct. This subroutine is for use when a packet capture device is opened using the **pcap_open_live** subroutine.

Parameters

Item	Description
p	Points to a packet capture descriptor as returned by the **pcap_open_live** subroutine.
ps	Points to the **pcap_stat** struct that will be filled in with the packet capture statistics.

Return Values

On successful completion, the **pcap_stats** subroutine fills in *ps* and returns 0. If the **pcap_stats** subroutine is unsuccessful, -1 is returned. In this case, the error text can be obtained with the **pcap_perror** subroutine or the **pcap_geterr** subroutine.

pcap_strerror Subroutine

Purpose

Obtains the error message indexed by *error*.

Library

pcap Library (**libpcap.a**)

Syntax

```
#include <pcap.h>

char *pcap_strerror(int  error);
```

Description

Lookup the error message indexed by *error.* The possible values of *error* correspond to the values of the *errno* global variable. This function is equivalent to the **strerror** subroutine.

Parameters

Item	Description
error	Specifies the key to use in obtaining the corresponding error message. The error message is taken from the system's **sys_errlist**.

Return Values

The **pcap_strerror** subroutine returns the appropriate error message from the system error list.

pclose Subroutine

Purpose

Closes a pipe to a process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
int pclose ( Stream)
FILE *Stream;
```

Description

The **pclose** subroutine closes a pipe between the calling program and a shell command to be executed. Use the **pclose** subroutine to close any stream you opened with the **popen** subroutine. The **pclose** subroutine waits for the associated process to end, and then returns the exit status of the command.

Attention: If the original processes and the **popen** process are reading or writing a common file, neither the **popen** subroutine nor the **pclose** subroutine should use buffered I/O. If they do, the results are unpredictable.

Avoid problems with an output filter by flushing the buffer with the **fflush** subroutine.

Parameter

Item	Description
Stream	Specifies the **FILE** pointer of an opened pipe.

Return Values

The **pclose** subroutine returns a value of -1 if the *Stream* parameter is not associated with a **popen** command or if the status of the child process could not be obtained. Otherwise, the value of the termination status of the command language interpreter is returned; this will be 127 if the command language interpreter cannot be executed.

Error Codes

If the application has:

- Called the **wait** subroutine,
- Called the **waitpid** subroutine with a process ID less than or equal to zero or equal to the process ID of the command line interpreter,
- Masked the SIGCHILD signal, or
- Called any other function that could perform one of the steps above, and

one of these calls caused the termination status to be unavailable to the **pclose** subroutine, a value of -1 is returned and the **errno** global variable is set to **ECHILD**.

pdmkdir Subroutine

Purpose

Creates or sets partitioned directories.

Syntax

```
#include <sys/secconf.h>
int pdmkdir (Path, Mode, Flag)
char *Path;
mode_t Mode;
int Flag;
```

Description

The pdmkdir subroutine creates a new partitioned directory or changes the type of the directory.

The process must be in real mode for the pdmkdir subroutine to succeed.

To run the pdmkdir subroutine, the PDMKDIR authorization is required to override the Discretionary Access Control (DAC), the Mandatory Access Control (MAC), and the Mandatory Integrity Control (MIC)

restrictions. Otherwise, the pdmkdir function can be used by the non-PDMKDIR-authorized users subject to the DAC, MAC, and MIC restrictions.

The nested partitioned directory is not supported by this subroutine because there is no advantage of having nested partitioned directory.

Parameters

Item	Description
Path	Specifies the name of the directory to be created or to be modified.
Mode	Specified the mask for the *read*, *write*, and *execute* flags for owners, group, and others. The *Mode* parameter specifies directory permissions and attributes.
Flag	Specifies the function to be performed by the pdmkdir subroutine. The *flag* parameter can be one of the following values: **MKPDIR** Creates a partitioned directory. **SETPDIR** Sets a directory to partitioned directory. The existing subdirectories do not become partitioned subdirectories and the existing file objects in this directory are not accessible in virtual mode.

Return Values

Upon successful completion, the pdmkdir subroutine returns a value of zero. Otherwise, it returns a value of nonzero.

Files

The sys/secconf.h file.

perfstat_bridgedadapters Subroutine

Purpose

Retrieves the underlying physical or virtual adapter statistics of the associated Shared Ethernet Adapter (SEA) adapter.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_bridgedadapters (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_netadapter_t * userbuff;
size_t sizeof_struct;int desired_number;
```

Description

The **perfstat_bridgedadapters** subroutine retrieves one or more SEA children adapter usage statistics.

The same function can also be used to retrieve the number of available sets of SEA children adapter statistics.

To get one or more sets of SEA adapter usage metrics, set the *name* parameter to the name of the SEA adapter for which the statistics are to be collected, and set the *desired_number* parameter. The valid SEA adapter name must be passed to the *name* parameter. The *userbuff* parameter must always point to the memory area that is big enough to contain the number of **perfstat_netadapter_t** structures that this subroutine is to copy. Upon return, the *name* parameter is set to either the name of the next SEA children adapter, or to the quotation marks (" ") after all of the structures are copied.

To retrieve the number of available sets of SEA children adapter usage metrics, pass the valid SEA name and set the *userbuff* parameter to the value of null, and the *desired_number* parameter to the value of zero. The returned value is the number of available sets.

Parameters

Item	Description
name	Contains the valid SEA adapter name. For example: ent0, ent1.
userbuff	Points to the memory that is to be filled with one or more **perfstat_netadapter_t** structures.
sizeof_struct	Specifies the size of the **perfstat_netadapter_t** structure.
desired_number	Specifies the number of **perfstat_netadapter_t** structures to copy to the *userbuff* parameter.

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_bridgedadapters** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient
ENOMEM	The default length of the string is too short.
ENOMSG	The dictionary is not accessible.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_cluster_disk Subroutine

Purpose

Retrieves the disk details of the cluster nodes.

Library

```
perfstat library (libperfstat.a)
```

Syntax

```
#include <libperfstat.h>

int perfstat_cluster_disk( name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_disk_data_t *userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_cluster_disk** subroutine returns the list of disks in a `perfstat_disk_data_t` structure.

The **perfstat_cluster_disk** subroutine must be called only after you enable the cluster statistics collection by using the following `perfstat` API call:

```
perfstat_config(PERFSTAT_ENABLE | PERFSTAT_CLUSTER_STATS, NULL)
```

The cluster statistics collection must be disabled after you get the list of disks by using the following `perfstat` API call:

```
perfstat_config(PERFSTAT_DISABLE | PERFSTAT_CLUSTER_STATS, NULL)
```

To identify the total number of cluster disks in a specific node (in which the current node is participating), the following criteria must be specified:

- The node name must be specified in the *name* parameter.
- The *userbuff* parameter must be set to NULL.
- The *desired_number* parameter must be set to 0.

To obtain the list of cluster disks in a specific node, the *userbuff* parameter and the *desired_number* parameter must be used.

Parameters

name.nodename or name.spec
Specifies the node name or the node ID for which the data must be returned.

userbuff
Specifies the memory area that must be filled with the `perfstat_disk_data_t` structure.

sizeof_userbuff
Specifies the size of the `perfstat_disk_data_t` structure.

desired_number
Specifies the number of structures to be returned.

Return values

The number of filled structures is returned upon successful completion. If unsuccessful, a value of -1 is returned and the *errno* global variable is set.

Error codes

The **perfstat_cluster_disk** subroutine fails because of one of the following errors:

EINVAL
One of the parameters is not valid.

ENOENT
The cluster statistics collection is not enabled by using the **perfstat_config** subroutine, the cluster statistics collection is not supported, or the specified node cannot be found.

Files

The `libperfstat.h` file defines standard macros, data types, and subroutines.

perfstat_cpu Subroutine

Purpose

Retrieves individual logical processor usage statistics.

Library

perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_cpu (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_cpu_t * userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_cpu** subroutine retrieves one or more individual processor usage statistics. The same function can be used to retrieve the number of available sets of logical processor statistics.

To get one or more sets of processor usage metrics, set the *name* parameter to the name of the first processor for which statistics are desired, and set the *desired_number* parameter. To start from the first processor, set the *name* parameter to "". The *userbuff* parameter must always point to a memory area big enough to contain the desired number of **perfstat_cpu_t** structures that will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next processor, or to "" after all structures have been copied.

To retrieve the number of available sets of processor usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The returned value will be the number of available sets.

This number represents the number of logical processors for which statistics are available. In a dynamic logical partitioning (DLPAR) environment, this number is the highest logical index of an online processor since the last reboot. See the Perfstat API article in Performance Tools and APIs Technical Reference for more information on the **perfstat_cpu** subroutine and DLPAR.

The SPLPAR environments virtualize physical processors. To help accurately measure the resource use in a virtualized environment, the POWER5 family of processors implements a register PURR (Processor Utilization Resource Register) for each core. The PURR is a 64-bit counter with the same units as the timebase register and tracks the real physical processor resource used on a per-thread or per-partition level. The PURR registers are not compatible with previous global counters (user, system, idle and wait fields) returned by the `perfstat_cpu` and the `perfstat_cpu_total` subroutines. All data consumers requiring processor utilization must be modified to support PURR-based computations as shown in the example for the **perfstat_partition_total** interface under Perfstat API programming.

This subroutine returns only global processor statistics inside a workload partition (WPAR).

Parameters

Item	Description
name	Contains either "", FIRST_CPU, or a name identifying the first logical processor for which statistics are desired. Logical processor names are: `cpu0, cpu1,...` To provide binary compatibility with previous versions of the library, names like *proc0, proc1, ...* will still be accepted. These names will be treated as if their corresponding *cpuN* name was used, but the names returned in the structures will always be names starting with *cpu*.
userbuff	Points to the memory area that is to be filled with one or more **perfstat_cpu_t** structures.
sizeof_struct	Specifies the size of the **perfstat_cpu_t** structure: `sizeof(perfstat_cpu_t)`.
desired_number	Specifies the number of **perfstat_cpu_t** structures to copy to *userbuff*.

Return Values

Unless the **perfstat_cpu** subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_cpu** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_cpu_rset Subroutine

Purpose

Retrieves the processor use statistics of resource set (rset)

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_cpu_rset (name, userbuff, sizeof_userbuff, desired_number)
perfstat_id_wpar_t * name;
perfstat_cpu_t * userbuff;
size_t sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_cpu_rset** subroutine returns the use statistics of the processors that belong to the specified resource set (rset).

To get the statistics of the processors that are in the resource set, specify the name or ID of the WPAR, or the rset handle for the WPAR name. If the name or ID of the WPAR is specified, the associated rset is taken. The *userbuff* parameter must be allocated, and the *desired_number* parameter must be the number of processors in the rset. When this subroutine is called inside a WPAR, the *name* parameter must be specified as NULL.

Parameters

Item	Description
name	Defines the WPAR name or WPAR ID. If the subroutine is called from WPAR, the value of the *name* parameter is null.
userbuff	Points to the memory area that is to be filled with the **perfstat_wpar_total_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_wpar_total_t** structure.
desired_number	Specifies the number of **perfstat_wpar_total_t** structures to copy to *userbuff*. The value of this parameter must be set to one.

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_cpu_rset** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_cpu_total_rset Subroutine

Purpose

Retrieves the processor use statistics of resource set (rset)

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_cpu_total_rset (name, userbuff, sizeof_userbuff, desired_number)
perfstat_id_wpar_t * name;
```

```
perfstat_cpu_total_rset_t * userbuff;
size_t sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_cpu_total_rset** subroutine returns the total use statistics of the processors that belong to the specified resource set (rset).

To get the statistics of the processor use by the rset, specify the WPAR ID. The *userbuff* parameter must be allocated, and the *desired_number* parameter must be set. When this subroutine is called inside a WPAR, the *name* parameter must be specified as NULL.

Parameters

Item	Description
name	Defines the WPAR name or the WPAR ID. If the subroutine is called from WPAR, the value of the *name* parameter is null.
userbuff	Points to the memory area that is to be filled with the **perfstat_cpu_total_rset_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_cpu_total_rset_t** structure.
desired_number	Specifies the number of **perfstat_cpu_total_rset_t** structures to copy to *userbuff*. The value of this parameter must be set to one.

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_cpu_rset** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient
ENOMEM	The default length of the string is too short.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_cpu_total_wpar Subroutine

Purpose

Retrieves workload partition (WPAR) processor use statistics

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_cpu_total_wpar ( name, userbuff, sizeof_userbuff, desired_number )
perfstat_id_wpar_t *name;
perfstat_cpu_total_wpar_t *userbuff;
size_t sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_cpu_total_wpar** subroutine returns workload partition (WPAR) processor use statistics in a **perfstat_cpu_total_wpar_t** structure.

To get statistics of any particular WPAR from global environment, the WPAR ID or the WPAR name must be specified in the *name* parameter. The *userbuff* parameter must be allocated and the *desired_number* parameter must be set to the value of one. When this subroutine is called inside a WPAR, the *name* parameter must be set to NULL.

Parameters

Item	Description
name	Specifies the WPAR ID or WPAR name. It is NULL if the subroutine is called from WPAR.
userbuff	Points to the memory area that is to be filled with the **perfstat_cpu_total_wpar_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_cpu_total_wpar_t** structure.
desired_number	Specifies the number of structures to return. The value of this parameter must be set to the value of one.

Return Values

Upon successful completion, the number of structures filled is returned. If unsuccessful, a value of -1 is returned, and the **errno** global variable is set.

Error Codes

The **perfstat_cpu_total_wpar** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	The memory is not sufficient.
ENOMEM	The default length of the string is too short.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_cpu_total Subroutine

Purpose

Retrieves global processor usage statistics.

Library

Perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_cpu_total (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_cpu_total_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_cpu_total** subroutine returns global processor usage statistics in a **perfstat_cpu_total_t** structure.

To get statistics that are global to the whole system, the *name* parameter must be set to NULL, the *userbuff* parameter must be allocated, and the *desired_number* parameter must be set to 1.

The **perfstat_cpu_total** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

The SPLPAR environments virtualize physical processors. To help accurately measure the resource used in a virtualized environment, the POWER5 family of processors implements a register PURR (Processor Utilization Resource Register) for each core. The PURR is a 64-bit counter with the same units as the timebase register and tracks the real physical processor resource used on a per-thread or per-partition level. The PURR registers are not compatible with previous global counters (user, system, idle and wait fields) returned by the `perfstat_cpu` and the `perfstat_cpu_total` subroutines. All data consumers requiring processor use must be modified to support PURR-based computations as shown in the example for the **perfstat_partition_total** interface under Perfstat API programming.

This subroutine returns only global processor statistics inside a workload partition (WPAR).

Parameters

Item	Description
name	Must set to NULL.
userbuff	Points to the memory area that is to be filled with the **perfstat_cpu_total_t** structure.
sizeof_struct	Specifies the size of the **perfstat_cpu_total_t** structure: `sizeof(perfstat_cpu_total_t)`.
desired_number	Must set to 1.

Return Values

Upon successful completion, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_cpu_total** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Item	Description
EFAULT	Insufficient memory.
ENOMEM	The string default length is too short.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_cluster_total Subroutine

Purpose

Retrieves cluster statistics

Library

perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_cluster_total  ( name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_cluster_total_t *userbuff;
int sizeof_userbuff;int desired_number;
```

Description

The **perfstat_cluster_total** subroutine returns the cluster statistics in a **perfstat_cluster_total_t** structure.

The **perfstat_cluster_total** subroutine should be called only after enabling cluster statistics collection by using the following perfstat API call: **perfstat_config(PERFSTAT_ENABLE | PERFSTAT_CLUSTER_STATS, NULL)** system call.

The cluster statistics collection must be disabled after collecting the cluster statistics by using the following perfstat API call: **perfstat_config(PERFSTAT_DISABLE | PERFSTAT_CLUSTER_STATS, NULL)**.

To get the statistics of any particular cluster (in which the current node is a cluster member) the cluster name must be specified in the *name* parameter. The *userbuff* parameter must be allocated. The *desired_number* parameter must be set to one.

Note: The cluster name should be one of the clusters in which the current node (in which the **perfstat** API call is run) is a cluster member.

Parameters

Item	Description
name.nodenamename.spec	Specifies the cluster name. Specifies the Cluster ID specifier. Should be set to CLUSTERNAME.
userbuff	Specifies the memory area that is to be filled with the **perfstat_cluster_total_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_cluster_total_t** structure.

Item	Description
desired_number	Specifies the number of structures to be returned. The value of this parameter must be set to one.

Return Values

Upon successful completion, the number of structures filled is returned. This will always be 1.

If unsuccessful, a value of -1 is returned, and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
ENOENT	Either cluster statistics collection is not enabled using the **perfstat_config** subroutine or the cluster statistics collection is not supported.
ENOSPC	The **ENOSPC** error code is set if either of the following cases occur: • If the **userbuff->node_data** is not NULL and initialized with insufficient memory (less than the total number of nodes in the cluster). • If **userbuff->disk_data** is not NULL and initialized with insufficient memory (less than the total number of disks in the cluster). Upon return, **userbuff->num_nodes** and **userbuff->num_disks** are initialized with the total number of nodes and disks respectively so that the user can reallocate sufficient memory and call the interface again.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_disk Subroutine

Purpose

Retrieves individual disk usage statistics.

Library

Perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_disk (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
```

```
perfstat_disk_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_disk** subroutine retrieves one or more individual disk usage statistics. The same function can also be used to retrieve the number of available sets of disk statistics.

To get one or more sets of disk usage metrics, set the *name* parameter to the name of the first disk for which statistics are desired, and set the *desired_number* parameter. To start from the first disk, specify "" or FIRST_DISK as the *name*. The *userbuff* parameter must always point to a memory area big enough to contain the desired number of **perfstat_disk_t** structures that will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next disk, or to "" after all structures have been copied.

To retrieve the number of available sets of disk usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The returned value will be the number of available sets.

The **perfstat_disk** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

To improve system performance, the collection of disk input and output statistics is disabled by default in current releases of AIX.

To enable the collection of this data, run:

```
chdev -l sys0 -a iostat=true
```

To display the current setting, run:

```
lsattr -E -l sys0 -a iostat
```

Another way to enable the collection of the disk input and output statistics is to use the `sys_parm` API and the SYSP_V_IOSTRUN flag:

To get the current status of the flag, run the following:

```
struct vario var;
sys_parm(SYSP_GET,SYSP_V_IOSTRUN, &var);
```

To set the flag, run the following:

```
struct vario var;
var.v.v_iostrun.value=1; /* 1 to set & 0 to unset */
sys_parm(SYSP_SET,SYSP_V_IOSTRUN, &var);
```

Parameters

Item	Description
name	Contains either "", FIRST_DISK, or a name identifying the first disk for which statistics are desired. For example: `hdisk0, hdisk1, ...`
userbuff	Points to the memory area to be filled with one or more **perfstat_disk_t** structures.
sizeof_struct	Specifies the size of the **perfstat_disk_t** structure: `sizeof(perfstat_disk_t)`
desired_number	Specifies the number of **perfstat_disk_t** structures to copy to *userbuff*.

Return Values

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_disk** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Insufficient memory.
ENOMEM	The string default length is too short.
ENOMSG	Cannot access the dictionary.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_cpu_util Subroutine

Purpose

Calculates central processing unit utilization.

Library

perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_cpu_util (cpustats, userbuff, sizeof_userbuff, desired_number)
perfstat_rawdata_t * cpustats;
perfstat_cpu_util_t * userbuff;
int  sizeof_userbuff ;
int  desired_number ;
```

Description

The **perfstat_cpu_util** subroutine calculates the CPU utilization-related metrics for the current and the previous values passed to the **perfstat_rawdata_t** data structure. Both the system utilization and the per CPU utilization values can be obtained, using the same API, by mentioning the type field of the **perfstat_rawdata_t** data structure as **UTIL_CPU_TOTAL** or **UTIL_CPU**. The **UTIL_CPU_TOTAL** and **UTIL_CPU** are the macros, which can be referred to in the definition of the **perfstat_rawdata_t** data structure. If the attributes *name* and *userbuff* are set to NULL, and the *sizeof_userbuff* parameter is set to zero, the size of the current version of the **perfstat_cpu_util_t** structure is returned. If the *desired_elements* parameter is set to zero, the number of current elements, from the *cpustats* parameter, are returned.

Parameters

Item	Description
cpustats	Calculates the utilization-related metrics from the current and the previous values. The *cpustats* parameter is of the type **perfstat_rawdata_t**. The **curstat** and the **prevstat** attributes, points to the **perfstat_cpu_util_t** data structure. **Note:** To calculate the partition level CPU utilization, set the *cpustats* parameter to **UTIL_CPU_TOTAL** . For the individual CPU utilization, set the *cpustats* parameter to **UTIL_CPU**. The ID of the individual CPU can also be specified in the *cpustats* parameter if utilization to be calculated applies only to a specific CPU.
userbuff	Specifies the memory area that is to be filled with one or more **perfstat_cpu_util_t** structures.
sizeof_userbuff	Specifies the size of the **perfstat_cpu_util_t** structure. **Note:** To obtain the size of the latest version of **perfstat_cpu_util_t** structure, set the *sizeof_userbuff* parameter to 0, and set the *name* and *userbuff* parameters to NULL.
desired_number	Specifies the number of **perfstat_cpu_util_t** structures to copy to the *userbuff* parameter.

Return Values

Unless the **perfstat_cpu_util** subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_cpu_util** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_diskadapter Subroutine

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Purpose

Retrieves individual disk adapter usage statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>


int perfstat_diskadapter (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
 perfstat_diskadapter_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_diskadapter** subroutine retrieves one or more individual disk adapter usage statistics. The same function can be used to retrieve the number of available sets of adapter statistics.

To get one or more sets of disk adapter usage metrics, set the *name* parameter to the name of the first disk adapter for which statistics are desired, and set the *desired_number* parameter. To start from the first disk adapter, set the *name* parameter to "" or FIRST_DISKADAPTER. The *userbuff* parameter must point to a memory area big enough to contain the desired number of **perfstat_diskadapter_t** structures which will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next disk adapter, or to "" if all structures have been copied.

To retrieve the number of available sets of disk adapter usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The returned value will be the number of available sets.

The **perfstat_diskadapter** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

To improve system performance, the collection of disk input/output statistics is disabled by default in current releases of AIX.

To enable the collection of this data, use:

```
chdev -l sys0 -a iostat=true
```

To display the current setting, use:

```
lsattr -E -l sys0 -a iostat
```

Another way to enable the collection of the disk input/output statistics is to use the `sys_parm` API and the SYSP_V_IOSTRUN flag:

To get the current status of the flag:

```
struct vario var;
sys_parm(SYSP_GET,SYSP_V_IOSTRUN, &var);
```

To set the flag:

```
struct vario var;
var.v.v_iostrun.value=1; /* 1 to set & 0 to unset */
sys_parm(SYSP_SET,SYSP_V_IOSTRUN, &var);
```

Parameters

Item	Description
name	Contains either "", FIRST_DISKADAPTER, or a name identifying the first disk adapter for which statistics are desired. For example: `scsi0, scsi1, ...`

Item	Description
userbuff	Points to the memory area to be filled with one or more **perfstat_diskadapter_t** structures.
sizeof_struct	Specifies the size of the **perfstat_diskadapter_t** structure: `sizeof(perfstat_diskadapter_t)`
desired_number	Specifies the number of **perfstat_diskadapter_t** structures to copy to *userbuff*.

Return Values

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_diskadapter** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Insufficient memory.
ENOMEM	The string default length is too short.
ENOMSG	Cannot access the dictionary.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_diskpath Subroutine

Purpose

Retrieves individual disk path usage statistics.

Library

Perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_diskpath (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_diskpath_t *userbuff
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_diskpath** subroutine retrieves one or more individual disk path usage statistics. The same function can also be used to retrieve the number of available sets of disk path statistics.

To get one or more sets of disk path usage metrics, set the *name* parameter to the name of the first disk path for which statistics are desired, and set the *desired_number* parameter. To start from the first disk path, specify "" or FIRST_DISKPATH as the *name* parameter. To start from the first path of a specific disk,

set the *name* parameter to the diskname. The *userbuff* parameter must always point to a memory area big enough to contain the desired number of **perfstat_diskpath_t** structures that will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next disk path, or to "" after all structures have been copied.

To retrieve the number of available sets of disk path usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The number of available sets is returned.

The **perfstat_diskpath** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

To improve system performance, the collection of disk input and output statistics is disabled by default in current releases of AIX.

To enable the collection of this data, run:

```
chdev -l sys0 -a iostat=true
```

To display the current setting, run:

```
lsattr -E -l sys0 -a iostat
```

Another way to enable the collection of the disk input and output statistics is to use the `sys_parm` API and the SYSP_V_IOSTRUN flag:

To get the current status of the flag, run the following:

```
struct vario var;
sys_parm(SYSP_GET,SYSP_V_IOSTRUN, &var);
```

To set the flag, run the following:

```
struct vario var;
var.v.v_iostrun.value=1; /* 1 to set & 0 to unset */
sys_parm(SYSP_SET,SYSP_V_IOSTRUN, &var);
```

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Parameters

Item	Description
name	Contains either "", FIRST_DISKPATH, a name identifying the first disk path for which statistics are desired, or a name identifying a disk for which path statistics are desired. For example: `hdisk0_Path2, hdisk1_Path0, ... or hdisk5 (equivalent to hdisk5_Path`*`firstpath`*`)`
userbuff	Points to the memory area to be filled with one or more **perfstat_diskpath_t** structures.
sizeof_struct	Specifies the size of the **perfstat_diskpath_t** structure: `sizeof(perfstat_diskpath_t)`
desired_number	Specifies the number of **perfstat_diskpath_t** structures to copy to *userbuff*.

Return Values

Unless the function is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_diskpath** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Insufficient memory.
ENOMEM	The string default length is too short.
ENOMSG	Cannot access the dictionary.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_disk_total Subroutine

Purpose

Retrieves global disk usage statistics.

Library

Perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_disk_total (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_disk_total_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_disk_total** subroutine returns global disk usage statistics in a **perfstat_disk_total_t** structure.

To get statistics that are global to the whole system, the *name* parameter must be set to NULL, the *userbuff* parameter must be allocated, and the *desired_number* parameter must be set to 1.

The **perfstat_disk_total** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

To improve system performance, the collection of disk input and output statistics is disabled by default in current releases of AIX.

To enable the collection of this data, run:

```
chdev -l sys0 -a iostat=true
```

To display the current setting, run:

```
lsattr -E -l sys0 -a iostat
```

Another way to enable the collection of the disk input and output statistics is to use the `sys_parm` API and the SYSP_V_IOSTRUN flag:

To get the current status of the flag, run the following:

```
struct vario var;
sys_parm(SYSP_GET,SYSP_V_IOSTRUN, &var);
```

To set the flag, run the following:

```
struct vario var;
var.v.v_iostrun.value=1; /* 1 to set & 0 to unset */
sys_parm(SYSP_SET,SYSP_V_IOSTRUN, &var);
```

Parameters

Item	Description
name	Must set to NULL.
userbuff	Points to the memory area that is to be filled with one or more **perfstat_disk_total_t** structures.
sizeof_struct	Specifies the size of the **perfstat_disk_total_t** structure: `sizeof(perfstat_disk_total_t)`
desired_number	Must set to 1.

Return Values

Upon successful completion, the number of structures that could be filled is returned. This is always 1. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_disk_total** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Insufficient memory.
ENOMEM	The string default length is too short.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_fcstat Subroutine

Purpose

Retrieves the statistics of a Fibre Channel (FC) adapter.

Library

Perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_fcstat (name, userbuff, sizeof_struct, desired_number)

perfstat_id_t *name;
perfstat_fcstat_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_fcstat** subroutine retrieves the statistics of one or more FC adapters. The same function is also used to retrieve the number of available FC adapter statistics.

To get one or more FC adapter statistics, specify the name of the first FC adapter for which you want the statistics by the **name** parameter and set the **desired_number** parameter accordingly. To start from the first FC adapter, set the **name** parameter to "" or *FIRST_FCADAPTER*. The **userbuff** parameter always points to a memory area that can contain the desired number of **perfstat_fcstat_t** structures that are copied by this function. On successful completion of the subroutine, the **name** parameter is set to the name of the next FC adapter or to "" after all the structures have been copied.

To retrieve the number of available FC adapter statistics, set the **name** and **userbuff** parameters to *NULL*, and the **desired_number** parameter to 0. The value returned is the number of available adapters.

Note:

For nonroot user, the values return by the **perfstat_fcstat** subroutine will always be zero for all listed fiber channel adapters.

Parameters

Item	Description
name	Specifies either "" or *FIRST_FCADAPTER*, or the name of the first network adapter for which statistics are required. For example, *fcs0* or *fcs1*.
userbuff	Points to the memory area that is to be filled with one or more **perfstat_fcstat_t** structures.
sizeof_struct	Specifies the size of the **perfstat_fcstat_t** structure.
desired_number	Specifies the number of **perfstat_fcstat_t** structures to copy to the **userbuff** pointer.

Return Values

On successful completion of the subroutine unless the function is used to retrieve the number of available structures, the number of structures filled is returned. If the subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Memory is not sufficient.
ENOMEM	The default length of the string is too short.

Item	Description
ENOMSG	Cannot access the dictionary.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_fcstat_wwpn Subroutine

Purpose

Retrieves the Fibre Channel (FC) adapter statistics for a worldwide port name (WWPN) ID.

Library

Perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_fcstat_wwpn (name, userbuff, sizeof_struct, desired_number)

perfstat_wwpn_id_t *name;
perfstat_fcstat_t *userbuff;
size_t sizeof_struct;
int desired_number;

typedef struct { /* structure element identifier */
char name[IDENTIFIER_LENGTH]; /* name of the fc adapter identifier */
u_longlong_t initiator_wwpn_name;  /* initiator, WWPN name */ }
perfstat_wwpn_id_t;
```

Description

The **perfstat_fcstat_wwpn** subroutine retrieves individual FC adapter statistics for a specified WWPN ID.

Note: The **perfstat_fcstat_wwpn** subroutine does not work for the nonroot user.

Parameters

Item	Description
name	Specifies the name of the FC adapter and the WWPN name, for which the statistics are captured. If it is set to **NULL**, the error message is displayed.
userbuff	Points to the memory area that is to be filled with the **perfstat_fcstat_t** structure.
sizeof_struct	Specifies the size of the **perfstat_fcstat_t** structure.
desired_number	Specifies the number of **perfstat_fcstat_t** structures that are copied to the **userbuff** pointer. The parameter is set to 1 for the **perfstat_fcstat_wwpn** subroutine.

Return Values

On successful completion of the subroutine unless the function is used to retrieve an available structure, a filled structure is returned. If the subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
ENOMEM	The default length of the string is too short.

perfstat_hfistat Subroutine

Purpose

Retrieves the Host Fabric Interface (HFI) performance statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_hfistat (name,userbuff,sizeof_userbuff,desired_number )
perfstat_id_t* name;
perfstat_hfistat_t* userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_hfistat** subroutine returns the HFI performance statistics that correspond to a specified Host Fabric Interface.

To get the number of available HFI in the system, the *name* parameter and the *userbuff* parameter must be specified as NULL, *sizeof_userbuff* must equal the **sizeof (perfstat_hfistat_t)** subroutine and the value of the *desired_number* parameter must be set to zero.

To get one or more sets of HFI performance metrics, set the *name* parameter to the name of the first HFI for which the statistics is desired, and set the *desired_number* parameter. The *userbuff* parameter must be allocated.

Note: A **perfstat_config()** query verifies if the HFI statistics collection is available.
perfstat_config(PERFSTAT_QUERY|PERFSTAT_HFISTATS, NULL);

Parameters

Item	Description
name	Contains either **FIRST_HFI**, or a name that identified the first HFI for which statistics is desired. For example: **hfi0** and **hfi1**.
userbuff	Points to the memory area to be filled with one or more **perfstat_hfistat_t** structures.
sizeof_userbuff	Specifies the size of the **perfstat_hfistat_t** structure (**sizeof (perfstat_hfistat_t)**).
desired_number	Specifies the number of **perfstat_hfistat_t** structures to copy to the **userbuff**.

Return Values

Unless the subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of **-1** is returned and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
ENOENT	HFI statistics collection is currently not available.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_hfistat_window Subroutine

Purpose

Retrieves Host Fabric Interface (HFI) window-based performance statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_hfistat_window (name,userbuff,sizeof_userbuff,desired_number)
perfstat_id_window_t* name;
perfstat_hfistat_window_t* userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_hfistat_window** subroutine returns window-based performance statistics of a Host Fabric Interface in a **perfstat_hfistat_window_t** structure.

To get the maximum number of windows of a HFI in the system, specify the HFI name in the *name* parameter. The *userbuff* parameter must be specified as NULL, the *sizeof_userbuff* must be equal to the **sizeof (perfstat_hfistat_window_t)** and the value of the *desired_number* parameter must be set to zero.

To get one or more sets of HFI window-based performance metrics, specify the Host Fabric Interface name in the *name* parameter and the first desired window number in the *windowid* parameter. Specify the number of Host Fabric Interface windows for which performance statistics are to be collected in the *desired_number* parameter. The *userbuff* parameter must be allocated.

Note: A **perfstat_config()** query verifies if the HFI statistics collection is available or not (**perfstat_config(PERFSTAT_QUERY|PERFSTAT_HFISTATS, NULL);**).

Parameters

Item	Description
name–>name	Specifies the Host Fabric Interface. For example: hfi0, hfi1, and so forth.

Item	Description
name->windowid	Specifies the first desired window ID. For example: 0, 1, 2, 3, and so forth.
userbuff	Points to the memory area that is to be filled with the **perfstat_hfistat_window_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_hfistat_window_t** structure.
desired_number	Specifies the number of structures to return.

Return Values

Unless the subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if the following are true:

Item	Description
EINVAL	One of the parameters is not valid.
ENOENT	The HFI statistics collection is not currently available.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_logicalvolume Subroutine

Purpose

Retrieves logical volume related metrics

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_logicalvolume (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_logicalvolume_t * userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_logicalvolume** subroutine retrieves one or more logical volume statistics. It can also be used to retrieve the number of available logical volume.

To get one or more sets of logical volume metrics, set the *name* parameter to the name of the first logical volume for which the statistics are to be collected, and set the *desired_number* parameter. To start from the first logical volume, specify the quotation marks (“”) or FIRST_LOGICALVOLUME as the name. The *userbuff* parameter must always point to the memory area that is big enough to contain the number of

perfstat_logicalvolume_t structures that this subroutine is to copy. Upon return, the *name* parameter is set to either the name of the next logical volume, or to "" after all of the structures are copied.

To retrieve the number of available sets of logical volume metrics, set the *name* parameter and the *userbuff* parameter to the value of null, and the *desired_number* parameter to the value of zero. The returned value is the number of available logical volumes.

Note: The **perfstat_config** must be called to enable the logical volume statistics collection. The **perfstat_logicalvolume** subroutine is not supported inside workload partitions.

Parameters

Item	Description
name	Contains the quotation marks (""), FIRST_LOGICALVOLUME, or the name indicating the logical volume for which the statistics is to be retrieved
userbuff	Points to the memory that is to be filled with the **perfstat_logicalvolume_t** structure
sizeof_struct	Specifies the size of the **perfstat_logicalvolume_t** structure
desired_number	Specifies the number of different logical volume statistics to be collected

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned.

Error Codes

The **perfstat_logicalvolume** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_memory_page Subroutine

Purpose

Retrieves usage statistics for multiple page sizes.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_memory_page ( psize, userbuff, sizeof_userbuff, desired_number )
perfstat_psize_t *psize;
perfstat_memory_total_wpar_t *userbuff;
size_t sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_memory_page** subroutine returns the statistics corresponding to the different page sizes.

To get the number of supported page sizes, the *psize* parameter and the *userbuff* parameter must be specified as NULL, and the value of the *desired_number* parameter must be set to zero.

To get the statistics for the supported page sizes, specify the page size in the *psize* parameter. The *desired_number* parameter specifies the number of different page size statistics to be collected. The *userbuff* parameter must be allocated.

Parameters

Item	Description
psize	Specifies the page size for which the statistics are to be collected.
userbuff	Points to the memory area that is to be filled with the **perfstat_memory_page_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_memory_page_t** structure.
desired_number	Specifies the number of different page size statistics to be collected.

Return Values

Upon successful completion the number of **perfstat_memory_page_t** structures that are filled is returned. If the specified page size is not used, the returned value is 0. For example, if a user specified 4K page size, the return value is 0 since the specified page size is not used.

If unsuccessful, a value of -1 is returned, and the **errno** global variable is set.

Error Codes

The **perfstat_memory_page** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_memory_page_wpar Subroutine

Purpose

Retrieves use statistics for multiple page size for workload partitions (WPAR)

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_memory_page_wpar ( name, psize, userbuff, sizeof_userbuff, desired_number )
perfstat_id_wpar_t *name;
perfstat_psize_t *psize;
perfstat_memory_total_wpar_t *userbuff;
```

```
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_memory_page_wpar** subroutine returns the page statistics for the WPAR in **perfstat_memory_page_wpar_t** structure.

To get the statistics of the particular page size, the name of the WPAR must be specified with the *psize* parameter, the *userbuff* parameter must be allocated, and the *desired_number* parameter must be set to the number of structures to be retrieved.

Parameters

Item	Description
name	Specifies the name or ID of a WPAR to get the memory page statistics of the particular WPAR. If the memory page size statistics belongs to the calling process need to be retrieved, the value of this parameter is null. When the subroutine is called inside a WPAR, only the value of null can be specified.
psize	Specifies the page size for which the statistics are to be collected.
userbuff	Points to the memory area that is to be filled with the **perfstat_memory_page_wpar_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_memory_page_wpar_t** structure.
desired_number	Specifies the number of different page size statistics to be collected.

Return Values

Upon successful completion, the number of structures filled is returned. The returned value is one.

If unsuccessful, a value of -1 is returned.

Error Codes

The **perfstat_memory_page_wpar** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_memory_total_wpar Subroutine

Purpose

Retrieves workload partition (WPAR) memory use statistics

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_memory_total_wpar ( name, userbuff, sizeof_userbuff, desired_number )
perfstat_id_wpar_t *name;perfstat_memory_total_wpar_t *userbuff;
size_t sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_memory_total_wpar** subroutine returns workload partition (WPAR) memory use statistics in the **perfstat_memory_total_wpar_t** structure.

To get statistics of any particular WPAR from global environment, the WPAR ID or the WPAR name must be specified in the *name* parameter. The *userbuff* parameter must be allocated and the *desired_number* parameter must be set to the value of one. When this subroutine is called inside a WPAR, the *name* parameter must be set to NULL.

Parameters

Item	Description
name	Specifies the WPAR ID or the WPAR name. It is NULL if the subroutine is called from WPAR.
userbuff	Points to the memory area that is to be filled with the **perfstat_memory_total_wpar_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_memory_total_wpar_t** structure.
desired_number	Specifies the number of structures to return.

Return Values

Upon successful completion, the number of structures filled is returned. The returned value is one.

If unsuccessful, a value of -1 is returned, and the **errno** global variable is set.

Error Codes

The **perfstat_memory_total_wpar** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_memory_total Subroutine

Purpose

Retrieves global memory usage statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_memory_total (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_memory_total_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_memory_total** subroutine returns global memory usage statistics in a **perfstat_memory_total_t** structure.

To get statistics that are global to the whole system, the *name* parameter must be set to NULL, the *userbuff* parameter must be allocated, and the *desired_number* parameter must be set to 1.

This subroutine returns only global processor statistics inside a workload partition (WPAR).

Parameters

Item	Description
name	Must be set to NULL.
userbuff	Points to the memory area that is to be filled with the **perfstat_memory_total_t** structure.
sizeof_struct	Specifies the size of the **perfstat_memory_total_t** structure: `sizeof(perfstat_memory_total_t)`.
desired_number	Must be set to 1.

Return Values

Upon successful completion, the number of structures filled is returned. This will always be 1. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_memory_total** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_netadapter Subroutine

Purpose

Retrieves the statistics of a network adapter.

Library

Perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_netadapter (name, userbuff, sizeof_struct, desired_number)

perfstat_id_t *name;
perfstat_netadapter_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_netadapter** subroutine retrieves one or more individual network adapter statistics. The same function is also used to retrieve the number of available network adapter statistics.

To get one or more network adapter statistics, specify the **name** parameter to the name of the first network adapter for which statistics are desired, and set the **desired_number** parameter accordingly. To start from the first network adapter, set the **name** parameter to "" or *FIRST_NETADAPTER*. The **userbuff** parameter always points to a memory area that can contain the desired number of **perfstat_netadpater_t** structures that are copied by this function. On successful completion of the subroutine, the **name** parameter is set to the name of the next network adapter or to "" after all the structures were copied.

To retrieve the number of available network adapter statistics, set the **name** and **userbuff** parameters to *NULL*, and the **desired_number** parameter to 0. The value returned is the number of available adapters.

Parameters

Item	Description
name	Specifies either "" or *FIRST_NETADAPTER*, or the name of the first network adapter for which statistics are desired. For example, *ent0* or *ent1*.
userbuff	Points to the memory area that is to be filled with one or more **perfstat_netadapter_t** structures.
sizeof_struct	Specifies the size of the **perfstat_netadapter_t** structure.
desired_number	Specifies the number of **perfstat_netadapter_t** structures to copy to **userbuff**.

Return Values

On successful completion of the subroutine unless the function is used to retrieve the number of available structures, the number of structures filled is returned. If the subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Memory is not sufficient.
ENOMEM	The default length of the string is too short.
ENOMSG	Cannot access the dictionary.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_netbuffer Subroutine

Purpose

Retrieves network buffer allocation usage statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_netbuffer (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_netbuffer_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_netbuffer** subroutine retrieves statistics about network buffer allocations for each possible buffer size. Returned counts are the sum of allocation statistics for all processors (kernel statistics are kept per size per processor) corresponding to a buffer size.

To get one or more sets of network buffer allocation usage metrics, set the *name* parameter to the network buffer size for which statistics are desired, and set the *desired_number* parameter. To start from the first network buffer size, specify "" or FIRST_NETBUFFER in the *name* parameter. The *userbuff* parameter must point to a memory area big enough to contain the desired number of **perfstat_netbuffer_t** structures which will be copied by this function.

Upon return, the *name* parameter will be set to either the ASCII size of the next buffer type, or to "" if all structures have been copied. Only the statistics for network buffer sizes that have been used are returned. Consequently, there can be holes in the returned array of statistics, and the structure corresponding to allocations of size 4096 may directly follow the structure for size 256 (in case 512, 1024 and 2048 have not been used yet). The structure corresponding to a buffer size not used yet is returned (with all fields set to 0) when it is directly asked for by name.

To retrieve the number of available sets of network buffer usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The returned value will be the number of available sets.

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Parameters

Item	Description
name	Contains either "", FIRST_NETBUFFER, or the size of the network buffer in ASCII. It is a power of 2. For example: `32, 64, 128, ..., 16384`
userbuff	Points to the memory area to be filled with one or more **perfstat_netbuffer_t** structures.

Item	Description
sizeof_struct	Specifies the size of the **perfstat_netbuffer_t** structure: `sizeof(perfstat_netbuffer_t)`
desired_number	Specifies the number of **perfstat_netbuffer_t** structures to copy to *userbuff*.

Return Values

Upon successful completion, the number of structures which could be filled is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_netbuffer** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_netinterface Subroutine

Purpose

Retrieves individual network interface usage statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_netinterface (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_netinterface_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_netinterface** subroutine retrieves one or more individual network interface usage statistics. The same function can also be used to retrieve the number of available sets of network interface statistics.

To get one or more sets of network interface usage metrics, set the *name* parameter to the name of the first network interface for which statistics are desired, and set the *desired_number* parameter. To start from the first network interface, set the *name* parameter to "" or FIRST_NETINTERFACE. The *userbuff* parameter must always point to a memory area big enough to contain the desired number of **perfstat_netinterface_t** structures that will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next network interface, or to "" after all structures have been copied.

To retrieve the number of available sets of network interface usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The returned value will be the number of available sets.

The **perfstat_netinterface** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Parameters

Item	Description
name	Contains either "", FIRST_NETINTERFACE, or a name identifying the first network interface for which statistics are desired. For example; `en0, tr10, ...`
userbuff	Points to the memory area that is to be filled with one or more **perfstat_netinterface_t** structures.
sizeof_struct	Specifies the size of the **perfstat_netinterface_t** structure: `sizeof(perfstat_netinterface_t)`
desired_number	Specifies the number of **perfstat_netinterface_t** structures to copy to *userbuff*.

Return Values

Upon successful completion unless the function is used to retrieve the number of available structures, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_netinterface** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Insufficient memory.
ENOMEM	The string default length is too short.
ENOMSG	Cannot access the dictionary.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_netinterface_total Subroutine

Purpose

Retrieves global network interface usage statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_netinterface_total (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_netinterface_total_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_netinterface_total** subroutine returns global network interface usage statistics in a **perfstat_netinterface_total_t** structure.

To get statistics that are global to the whole system, the *name* parameter must be set to NULL, the *userbuff* parameter must be allocated, and the *desired_number* parameter must be set to 1.

The **perfstat_netinterface_total** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Parameters

Item	Description
name	Must be set to NULL.
userbuff	Points to the memory area that is to be filled with the **perfstat_netinterface_total_t** structure.
sizeof_struct	Specifies the size of the **perfstat_netinterface_total_t** structure: `sizeof(perfstat_netinterface_total_t)`.
desired_number	Must be set to 1.

Return Values

Upon successful completion, the number of structures filled is returned. This will always be 1. If unsuccessful, a value of -1 is returned and the **errno** variable is set.

Error Codes

The **perfstat_netinterface_total** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Insufficient memory.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_node Subroutine

Purpose

These subroutines retrieve the performance statistics of the subsystem type for a remote node. The list of subroutines are:

- perfstat_cpu_node
- perfstat_cpu_total_node
- perfstat_disk_node
- perfstat_disk_total_node
- perfstat_diskadapter_node
- perfstat_diskpath_node
- perfstat_fcstat_node
- perfstat_logicalvolume_node
- perfstat_memory_page_node
- perfstat_memory_total_node
- perfstat_netadapter_node
- perfstat_netbuffer_node
- perfstat_netinterface_node
- perfstat_netinterface_total_node
- perfstat_pagingspace_node
- perfstat_partition_total_node
- perfstat_protocol_node
- perfstat_tape_node
- perfstat_tape_total_node
- perfstat_volumegroup_node

Library

Perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_cpu_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_cpu_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_cpu_total_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_cpu_total_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_disk_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_disk_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_disk_total_node (name, userbuff, sizeof_userbuff, desired_number)
```

```
perfstat_id_node_t *name;
perfstat_disk_total_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_diskadapter_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_diskadapter_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_diskpath_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_diskpath_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_fcstat_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_fcstat_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_logicalvolume_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_logicalvolume_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_memory_page_node (name, psize, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_psize_t *psize;
perfstat_memory_page_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_memory_total_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_memory_total_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_netadapter_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_netadapter_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_netbuffer_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_netbuffer_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_netinterface_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_netinterface_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_netinterface_total_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_netinterface_total_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_pagingspace_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_pagingspace_t *userbuff;
int sizeof_userbuff;
```

```
int desired_number;

int perfstat_partition_total_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_partition_total_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_protocol_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_protocol_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_tape_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_tape_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_tape_total_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_tape_total_t *userbuff;
int sizeof_userbuff;
int desired_number;

int perfstat_volumegroup_node (name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_volumegroup_t *userbuff;
int sizeof_userbuff;
int desired_number
```

Description

These subroutines return the performance statistics of the remote node in their corresponding **perfstat_subsystem_t** structure.

All these subroutines are called only after the node or cluster statistics collection is enabled by calling the **perfstat_config** function:

perfstat_config (PERFSTAT_ENABLE | PERFSTAT_CLUSTER_STATS, NULL)

The node or cluster statistics collection is disabled after collecting the remote node data by calling the **perfstat_config** function:

perfstat_config (PERFSTAT_DISABLE | PERFSTAT_CLUSTER_STATS, NULL)

To get the statistics from any particular node in the cluster, specify the **Node name** value in the **name** parameter. The **userbuff** parameter must be allocated. The **desired number** parameter must be set.

Note: The remote node and the current node in which the **perfstat** API call runs belong to the same cluster.

The **perfstat_fcstat_node** subroutine does not work for the nonroot user.

Parameters

Item	Description
name.u.nodename	Specifies the node name.
name.spec	Specifies the node specifier.
name.name	Specifies the first component for which statistics is collected. For example, hdisk0, hdisk1, cpu0, and cpu1.

Item	Description
psize	Specifies the page size for which the statistics is collected.
userbuff	Points to the memory area that is to be filled with the **perfstat_<subsystem>_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_<subsystem>_t** structure.
desired_number	Specifies the number of structures to return.

Return Values

On successful completion of the subroutine, the number of available structures is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **perfstat_node** subroutine fails if one or more of the following are true:

Item	Description
EINVAL	One of the parameters are not valid.
ENOENT	Either the cluster statistics collection is not enabled using **perfstat_config()**, or the cluster statistics collection is not currently supported.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_node_list Subroutine

Purpose

Retrieves the list of nodes in a cluster.

Library

perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_node_list  ( name, userbuff, sizeof_userbuff, desired_number)

perfstat_id_node_t *name;
perfstat_node_t *userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_node_list** subroutine returns the list of nodes in a **perfstat_node_t** structure.

The **perfstat_node_list** subroutine should be called only after enabling cluster statistics collection by using the following perfstat API call: **perfstat_config(PERFSTAT_ENABLE | PERFSTAT_CLUSTER_STATS, NULL)**.

The cluster statistics collection must be disabled after collecting the node list by using the following perfstat API call: **perfstat_config(PERFSTAT_DISABLE | PERFSTAT_CLUSTER_STATS, NULL)**.

To obtain the total number of nodes in a cluster (in which the current node is participating), the cluster name must be specified in the *name* parameter, the *userbuff* parameter must be specified as NULL and the *desired_number* parameter must be specified as zero.

To obtain the list of nodes in a particular cluster (in which the current node is participating), the cluster name must be specified in the *name* parameter. The *userbuff* parameter must be allocated. The *desired_number* parameter must be set.

Note: The cluster name should be one of the clusters in which the current node (in which the perfstat API call is run) is participating.

Parameters

Item	Description
name.nodenamename.spec	Specifies the cluster name. Specifies the Cluster ID specifier. Should be set to CLUSTERNAME.
userbuff	Specifies the memory area that is to be filled with the **perfstat_node_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_node_t** structure.
desired_number	Specifies the number of structures to be returned.

Return Values

Unless the **perfstat_node_list** subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
ENOENT	Either cluster statistics collection is not enabled using **perfstat_config** or cluster statistics collection is currently not supported.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_pagingspace Subroutine

Purpose

Retrieves individual paging space usage statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>


int perfstat_pagingspace (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_pagingspace_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

This function retrieves one or more individual pagingspace usage statistics. The same functions can also be used to retrieve the number of available sets of paging space statistics.

To get one or more sets of paging space usage metrics, set the *name* parameter to the name of the first paging space for which statistics are desired, and set the *desired_number* parameter. To start from the first paging space, set the *name* parameter to "" or FIRST_PAGINGSPACE. In either case, *userbuff* must point to a memory area big enough to contain the desired number of **perfstat_pagingspace_t** structures which will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next paging space, or to "" if all structures have been copied.

To retrieve the number of available sets of paging space usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The number of available sets will be returned.

The **perfstat_pagingspace** subroutine retrieves information from the ODM database. This information is automatically cached into a dictionary which is assumed to be frozen once loaded. The **perfstat_reset** subroutine must be called to flush the dictionary whenever the machine configuration has changed.

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Parameters

Item	Description
name	Contains either "", FIRST_PAGINGSPACE, or a name identifying the first paging space for which statistics are desired. For example: `paging00, hd6, ...`
userbuff	Points to the memory area to be filled with one or more **perfstat_pagingspace_t** structures.
sizeof_struct	Specifies the size of the **perfstat_pagingspace_t** structure: `sizeof(perfstat_pagingspace_t)`
desired_number	Specifies the number of **perfstat_pagingspace_t** structures to copy to *userbuff*.

Return Values

Unless the **perfstat_pagingspace**subroutine is used to retrieve the number of available structures, the number of structures which could be filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_pagingspace** subroutine is unsuccessful if one of the following are true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_partial_reset Subroutine

Purpose

Empties part of the libperfstat configuration information cache or resets system minimum and maximum counters for disks.

Library

perfstat library (`libperfstat.a`)

Syntax

```
#include <libperfstat.h>

int perfstat_partial_reset (name, resetmask)
char * name;
u_longlong_t resetmask;
```

Description

The `perfstat_cpu_total`, `perfstat_disk`, `perfstat_diskadapter`, `perfstat_netinterface`, and `perfstat_pagingspace` subroutines return configuration information that is retrieved from the ODM database and automatically cached by the library. Other metrics provided by the LVM library and the `swapqry` subroutine are also cached for performance purpose.

The `perfstat_partial_reset` subroutine flushes some of this information cache and should be called whenever an identified part of the machine configuration has changed.

The `perfstat_partial_reset` subroutine can be used to reset a particular component (such as `hdisk0` or `en1`) when the *name* parameter is not NULL and the *resetmask* parameter contains only one bit. It can also be used to remove a whole category (such as disks or disk paths) from the cached information.

When the *name* parameter is NULL, the *resetmask* parameter can contain a combination of bits, such as `FLUSH_DISK|RESET_DISK_MINMAX|FLUSH_CPUTOTAL`.

Several bit masks are available for the *resetmask* parameter. The behavior of the function is as follows:

***resetmask* value**	**Action when *name* is NULL**	**Action when *name* is not NULL and a single resetmask is set**
FLUSH_CPUTOTAL	Flush *speed* and *description* in the `perfstat_cputotal_t` structure	An error is returned, and `errno` is set to `EINVAL`.
FLUSH_DISK	Flush *description*, *adapter*, *size*, *free*, and *vgname* in every `perfstat_disk_t` structure. Flush the list of disk adapters. Flush *size*, *free*, and *description* in every `perfstat_diskadapter_t` structure.	Flush *description*, *adapter*, *size*, *free*, and *vgname* in the specified `perfstat_disk_t` structure. Flush *adapter* in every `perfstat_diskpath_t` that matches the disk name followed by `_Path`. Flush *size*, *free*, and *description* of each `perfstat_diskadapter_t` that is linked to a path leading to this disk or to the disk itself.
RESET_DISK_ALL	Reset system resident all fields in every `perfstat_disk_t` structure.	An error is returned, and `errno` is set to `EINVAL`.

resetmask value	Action when *name* is NULL	Action when *name* is not NULL and a single resetmask is set
RESET_DISK_MINMAX	Reset system resident *min_rserv*, *max_rserv*, *min_wserv*, *max_wserv*, *wq_min_time* and *wq_max_time* in every `perfstat_disk_t` structure.	An error is returned, and `errno` is set to ENOTSUP.
FLUSH_DISKADAPTER	Flush the list of disk adapters. Flush *size*, *free*, and *description* in every `perfstat_diskadapter_t` structure. Flush *adapter* in every `perfstat_diskpath_t` structure. Flush *description* and *adapter* in every `perfstat_disk_t` structure.	Flush the list of disk adapters. Flush *size*, *free*, and *description* in the specified `perfstat_diskadapter_t` structure.
FLUSH_DISKPATH	Flush *adapter* in every `perfstat_diskpath_t` structure.	Flush *adapter* in the specified `perfstat_diskpath_t` structure.
FLUSH_PAGINGSPACE	Flush the list of paging spaces. Flush *automatic*, *type*, *lpsize*, *mbsize*, *hostname*, *filename*, and *vgname* in every `perfstat_pagingspace_t` structure.	Flush the list of paging spaces. Flush *automatic*, *type*, *lpsize*, *mbsize*, *hostname*, *filename*, and *vgname* in the specified `perfstat_pagingspace_t` structure.
FLUSH_NETINTERFACE	Flush *description* in every `perfstat_netinterface_t` structure.	Flush *description* in the specified `perfstat_netinterface_t` structure.

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Parameters

Item	Description
name	Contains a name identifying the component that metrics should be reset from the libperfstat cache. If this parameter is NULL, matches every component.
resetmask	The category of the component if the *name* parameter is not NULL. The available values are listed in the preceding table. In case the *name* parameter is NULL, the *resetmask* parameter can be a combination of bits.

Return Values

The `perfstat_partial_reset` subroutine returns a value of `0` upon successful completion. If unsuccessful, a value of `-1` is returned, and the `errno` global variable is set to the appropriate code.

Error Codes

Item	Description
`EINVAL`	One of the parameters is not valid.

Files

The `libperfstat.h` file defines standard macros, data types, and subroutines.

perfstat_partition_config Subroutine

Purpose

Retrieves operating system and partition related information.

Library

perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_partition_config (name, userbuff, sizeof_userbuff, desired_number)
perfstat_id_t * name;
perfstat_partition_config_t * userbuff;
int  sizeof_userbuff ;
int  desired_number ;
```

Description

The **perfstat_partition_config** subroutine returns the operating- system and partition-related information in a **perfstat_partition_config_t** structure. To retrieve statistics for the whole system, the *name* parameter must be set to NULL, the *userbuff* parameter must be allocated, and the *desired_number* parameter must be set to 1. If the *name* and *userbuff* parameters are set to NULL, and the *sizeof_userbuff* is set to 0, then the size of current version of the **perfstat_partition_config** data structure is returned.

Parameters

Item	Description
name	Points to the memory area to be filled with the **perfstat_partition_config_t** structure. This parameter must be set to NULL.
userbuff	Points to the memory area to be filled with the **perfstat_partition_config_t** data structure.
sizeof_userbuff	Specifies the size of the **perfstat_partition_config_t** structure: **sizeof(perfstat_partition_config_t)**. **Note:** To obtain the size of the latest version of **perfstat_partition_config_t**, set the *sizeof_userbuff* parameter to zero, and the *name* and *userbuff* parameters to NULL.
desired_number	This parameter must be set to 1.

Return Values

Upon successful completion, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_partition_config** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_partition_total Subroutine

Purpose

Retrieves global Micro-Partitioning usage statistics.

Library

perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
       int perfstat_partition_total(name, userbuff, sizeof_struct, desired_number)
       perfstat_id_t *name;
       perfstat_partition_total_t *userbuff;
       size_t sizeof_struct;
       int desired_number;
       u_longlong_t reserved_pages;
       u_longlong_t reserved_pagesize.
```

Description

The **perfstat_partition_total** subroutine returns global Micro-Partitioning usage statistics in a **perfstat_partition_total_t** structure. To retrieve statistics that are global to the whole system, the *name* parameter must be set to NULL, the *userbuff* parameter must be allocated, and the *desired_number* parameter must be set to 1.

This subroutine returns partition wide metrics inside a workload partition (WPAR).

Parameters

Item	Description
name	Must be set to NULL.
userbuff	Points to the memory area to be filled with the **perfstat_partition_total_t** structures.
sizeof_struct	Specifies the size of the **perfstat_partition_total_t** structure: `sizeof(perfstat_partition_total_t)`.
desired_number	Must be set to 1.
reserved_pagesize	Specifies the size of the pages for reserved memory. Not for use with **DR** operations.
reserved_pages	Specifies the number of pages of type *reserved_pagesize*. This information can be retrieved by calling **vmgetinfo**. Not for use with **DR** operations.

Return Values

Upon successful completion, the number of structures filled is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	Insufficient memory.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_protocol Subroutine

Purpose

Retrieves protocol usage statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>


int perfstat_protocol (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t *name;
perfstat_protocol_t *userbuff;
size_t sizeof_struct;
int desired_number;
```

Description

The **perfstat_protocol** subroutine retrieves protocol usage statistics such as ICMP, ICMPv6, IP, IPv6, TCP, UDP, RPC, NFS, NFSv2, NFSv3. To get one or more sets of protocol usage metrics, set the *name* parameter to the name of the first protocol for which statistics are desired, and set the *desired_number* parameter.

To start from the first protocol, set the *name* parameter to "" or FIRST_PROTOCOL. The *userbuff* parameter must point to a memory area big enough to contain the desired number of **perfstat_protocol_t** structures which will be copied by this function. Upon return, the *name* parameter will be set to either the name of the next protocol, or to "" if all structures have been copied.

To retrieve the number of available sets of protocol usage metrics, set the *name* and *userbuff* parameters to NULL, and the *desired_number* parameter to 0. The returned value will be the number of available sets.

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Parameters

Item	Description
name	Contains either "ip", "ipv6", "icmp", "icmpv6", "tcp", "udp", "rpc", "nfs", "nfsv2", "nfsv3", "", or FIRST_PROTOCOL.
userbuff	Points to the memory area to be filled with one or more **perfstat_protocol_t** structures.
sizeof_struct	Specifies the size of the **perfstat_protocol_t** structure: `sizeof(perfstat_protocol_t)`
desired_number	Specifies the number of **perfstat_protocol_t** structures to copy to *userbuff*.

Return Values

Upon successful completion, the number of structures which could be filled is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_protocol** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_process Subroutine

Purpose

Retrieves process utilization metrics.

Library

perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_process (name, userbuff, sizeof_userbuff, desired_elements)
perfstat_id_t * name;
perfstat_process_t * userbuff;
int  sizeof_userbuff ;
int  desired_number ;
```

Description

The **perfstat_process** subroutine is the interface for per process utilization metrics. The **perfstat_process** subroutine retrieves one or more process statistics to populate the **perfstat_process_t** data structure. If the*name* and *userbuff* parameters are specified as NULL, and the *desired_elements* parameter is stated as 0, the **perfstat_process** subroutine returns the number of active-processes, excluding the waiting processes. If the *name* and *userbuff* parameters are set to NULL, and the *sizeof_userbuff* parameter is set to 0, then the size of the current version of the **perfstat_process_t** data structure is returned.

Note: To improve performance, the collection of process scope disk statistics is disabled by default. To enable the collection of this data, enter the following command:

```
schedo -p -o proc_disk_stats=1
```

Parameters

Item	Description
name	Determines whether the statistics must be captured for all the processes or for a specific process. The *name* parameter, must be set to NULL to obtain the statistics for all processes. For a specific process, the process ID must be mentioned. **Note:** The process ID must be passed as a string. For example, to retrieve the statistics for a process with process ID 5478, the *name* parameter must be set to 5478.
userbuff	Points to the memory area that is to be filled with one or more **perfstat_process_t** data structures.
sizeof_userbuff	Specifies the size of the **perfstat_process_t** data structure. **Note:** To obtain the size of the latest version of the **perfstat_process_t** data structure, set the *sizeof_userbuff* parameter to 0, and *name* and *userbuff* parameter to NULL.
desired_elements	Specifies the number of **perfstat_process_t** data structures to copy to the *userbuff* parameter.

Return Values

Unless the **perfstat_process** subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_process** subroutine is unsuccessful if the following error code is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_process_util Subroutine

Purpose

Calculates process utilization metrics.

Library

perfstat library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_process (data, userbuff, sizeof_userbuff, desired_number)
perfstat_id_t * data;
perfstat_process_t * userbuff;
int  sizeof_userbuff ;
int  desired_number ;
```

Description

The **perfstat_process_util** subroutine provides the interface for process utilization metrics. The **perfstat_process** subroutine retrieves one or more process statistics to populate the **perfstat_process_t** data structure. The **perfstat_process_util** subroutine uses the current and previous values to calculate the utilization-related metrics. If the *name* and *userbuff* parameters are set to NULL, and the *sizeof_userbuff* parameter is set to 0, then the size of the current version of the **perfstat_process_t** data structure is returned. If the *desired_number* parameter is set to 0, the number of current elements, from the **perfstat_rawdata_t** data structure, is returned.

Parameters

Item	Description
data	Specifies that the *data* parameter is of the type **perfstat_rawdata_t**. The **perfstat_rawdata_t** data structure can take the current and the previous values to calculate the utilization-related metrics.
userbuff	Specifies the memory area to be filled with one or more **perfstat_process_t** data structures.
sizeof_userbuff	Specifies the size of the **perfstat_process_t** data structure. **Note:** To obtain the size of the latest version of the data structure **perfstat_process_t**, set the parameter *sizeof_userbuff* to 0, and the parameters *name* and *userbuff* to NULL.
desired_number	Specifies the number of the **perfstat_process_t** structures to copy to the *userbuff* parameter.

Return Values

Unless the **perfstat_process_util** subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_process_util** subroutine is unsuccessful if the following error code is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_processor_pool_util subroutine

Purpose

Calculates the metrics related to the processor pool utilization.

Library

```
perfstat library (libperfstat.a)
```

Syntax

```
#include <libperfstat.h>
```

```
int perfstat_processor_pool_util (perfstat_rawdata_t * data ,perfstat_processor_pool_util_t *
userbuff
int sizeoff_userbuff,
int desired_number);
```

Description

The **perfstat_processor_pool_util** subroutine calculates the metrics related to the processor pool utilization for the current and the previous values passed to the **perfstat_rawdata_t** data structure.

Pool utilization is calculated by specifying the **Type** field of the **perfstat_rawdata_t** data structure to **SHARED_POOL_UTIL**. The **SHARED_POOL_UTIL** is a macro which can be referred to in the definition of the **perfstat_rawdata_t** data structure.

Parameters

data
Calculates the metrics related to the processor pool utilization related from the current and previous values.

The *data* parameter belongs to the **perfstat_rawdata_t** data structure type. The **curstat** and the **prevstat** attributes points to the **perfstat_partition_total** data structure.

userbuff
Specifies the memory area that is to be filled with one or more **perfstat_processor_util_t** structure.

sizeof_userbuff
Specifies the size of the **perfstat_processor_util_t** structure.

desired_number
Specifies the number of **perfstat_processor_util_t** structures to copy to the **userbuff** parameter. The value needs to be set to 1.

Error Codes

The **perfstat_processor_pool_util** subroutine is unsuccessful if the following is true:

EINVAL
The value is set if one of the parameters is not valid.

EPERM
The value is set if the performance data collection is not enabled.

Return Values

If the **data** parameter is set to NULL and the **userbuff** parameter is also set to NULL and the **sizeof_userbuff** parameter is set to 0, size of the **perfstat_processor_pool_util_t** subroutine is returned.

Unless the **perfstat_processor_pool_util** subroutine is used to retrieve the number of available structures, the number of structures filled is returned upon successful completion. Otherwise, a value of -1 is returned and the **errno** global variable is set.

Note: The **perfstat_processor_pool_util** subroutine requires performance data collection to be enabled to return the processor pool values.

perfstat_reset Subroutine

Purpose

Empties libperfstat configuration information cache.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

void perfstat_reset (void)
```

Description

The **perfstat_cpu_total**, **perfstat_disk**, **perfstat_diskadapter**, **perfstat_netinterface**, and **perfstat_pagingspace** subroutines return configuration information retrieved from the ODM database and automatically cached by the library.

The **perfstat_reset** subroutine flushes this information cache and should be called whenever the machine configuration has changed.

This subroutine is not supported inside a workload partition (WPAR). It is not aware of a WPAR.

Files

The **libperfstat.h** defines standard macros, data types and subroutines.

perfstat_ssp Subroutine

Purpose

Retrieves the shared storage pool (SSP) statistics and disks and Virtual Target Devices (VTDs) which are associated with SSP.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_ssp (name, userbuff, sizeof_struct, desired_number, spp_flag)
perfstat_id_t * name;
perfstat_ssp_t * userbuff;
size_t sizeof_struct;
int desired_number;
ssp_flag_t spp_flag;
```

Description

The **perfstat_ssp** subroutine retrieves the shared storage pool (SSP) statistics.

To retrieve the number of available disks in the SSP, set the *name* and *userbuff* parameters to **NULL**, and the *desired_number* parameter to **0** and flag to **SSPDISK**.

To retrieve the number of available VTDs in the SSP, set the *name* and *userbuff* parameters to **NULL**, and the *desired_number* parameter to **0** and flag to **SSPVTD**.

Parameters

Item	Description
name	Must be set to NULL.
userbuff	Points to the memory area that is to be filled with the **perfstat_ssp_t** structure. Memory is allocated to the *userbuff* with the calculation (sizeof (perfstat_ssp_t) * returned_count), where **returned_count** is the value obtained by setting the *name* parameter and *userbuff* parameter to NULL and the *desired_number* parameter to zero.
sizeof_struct	Specifies the size of the **perfstat_ssp_t** structure.
desired_number	Must be set to 1.
spp_flag	Must be set to one of the following values: • SSPGLOBAL • SSPDISK • SSPVTD

Usage of the SSPGLOBAL flag

- When the **SSPGLOBAL** flag is invoked with the *name* and *userbuff* parameters set to **NULL**, the **perfstat_ssp** subroutine returns the number of SSPs available.
- When the **SSPGLOBAL** flag is invoked with enough space allocated to the *userbuff* parameter based on the return value of the previous call, the **perfstat_ssp** subroutine populates the SSP statistics.

Usage of the SSPDISK flag

- When the **SSPDISK** flag is invoked with the *name* and *userbuff* parameters set to **NULL**, the **perfstat_ssp** subroutine returns the number of disks associated with any SSP.
- When the **SSPDISK** flag is invoked with enough space allocated to the *userbuff* parameter based on the return value of the previous call, the **perfstat_ssp** subroutine populates the disk information with the cluster and pool name.

Usage of the SSPVTD flag

- When the **SSPVTD** flag is invoked with the *name* and *userbuff* parameters set to **NULL**, the **perfstat_ssp** subroutine returns the number of logical units associated with the SSP.
- When the **SSPVTD** flag is invoked with enough space allocated to the *userbuff* parameter based on the return value of the previous call, the **perfstat_ssp** subroutine populates the logical unit name, VTD name and type, and size utilization to the respective fields along with the cluster and pool name.

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned.

Error Codes

The **perfstat_ssp** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient
ENOMEM	The default length of the string is too short.
ENOMSG	The dictionary is not accessible.
ETIMEDOUT	The connection is timed out.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_ssp_ext Subroutine

Purpose

Retrieves the tier, failure group, physical volume, and node data that are associated with shared storage pool (SSP).

Syntax

```
#include <libperfstat.h>

int perfstat_ssp_ext (name, userbuff, sizeof_struct, desired_number, ssp_flag)
perfstat_ssp_id_t * name;
perfstat_ssp_t * userbuff;
size_t sizeof_struct;
int desired_number;
ssp_flag_t ssp_flag;
```

Description

The **perfstat_ssp_ext** subroutine retrieves the SSP statistics on the tier, failure group, and the physical volumes that belong to the failure group. This subroutine also retrieves nodes data that belong to the SSP.

To retrieve the number of tiers in the SSP, you must set the *name* and *userbuff* parameters to NULL, the *desired_number* parameter to 0, and the *ssp_flag* parameter to SSPTIER.

To retrieve the number of available failure groups in the SSP, you must set the *name* and *userbuff* parameters to NULL, the *desired_number* parameter to 0, and the *ssp_flag* parameter to SSPFG.

To retrieve the number of physical volumes that are associated with the SSP, you must set the *name* and *userbuff* parameters to NULL, the *desired_number* parameter to 0, and the *ssp_flag* parameter to SSPPV.

To retrieve the number of nodes that are associated with the SSP, you must set the *name* and *userbuff* parameters to NULL, the *desired_number* parameter to 0, and the *ssp_flag* parameter to SSPNODE.

To retrieve data that is specific to a tier, failure group, physical volume, or node, you must specify the *name* parameter.

To enable collection of these data, you must configure the cluster statistics collection by running the following subroutine:

```
perfstat_config(PERFSTAT_ENABLE | PERFSTAT_CLUSTER_STATS, NULL)
```

After the SSP data collection is complete, you must disable the cluster statistics collection by running the following subroutine:

```
perfstat_config(PERFSTAT_DISABLE | PERFSTAT_CLUSTER_STATS, NULL)
```

Note: The **perfstat_ssp_ext** subroutine can function only on Virtual I/O Server (VIOS).

Parameters

name
Filter for retrieving the tier, failure group, and physical volumes in a shared storage pool. The following filters are possible:

name->tier.name
Specifies the tier name for which data must be returned.

name->tier.id
Specifies the tier ID for which data must be returned.

name.fg.name
Specifies the failure group name for which data must be returned.

name->fg.id
Specifies the failure group ID for which data must be returned.

name->pv.name
Specifies the physical volume name for which data must be returned.

name->pv.id
Specifies the physical volume ID for which data must be returned.

name->name
Specifies the node name for which the data must be returned.

name->spec
Specifies the filter. The following values can be specified for this attribute:

PERFFILT_ID
Specifies that the filters are based on ID of a tier, failure group, or physical volume.

PERFFILT_NAME
Specifies that the filters are based on name of a tier, failure group, physical volume, or node.

Note: Both ID and name cannot be used at the same time.

PERFFILT_TIER
Specifies that the filters are specific to a tier. The filter can be based on tier ID or tier name. The **spec** attribute must be set accordingly.

PERFFILT_FG
Specifies that the filters are specific to a failure group. The filter can be based on failure group ID or failure group name. The **spec** attribute must be set accordingly.

PERFFILT_PV
Specifies that the filters are specific to a physical volume. The filter can be based on unique disk identifier (UDID) or physical volume name. The **spec** attribute must be set in both the cases.

PERFFILT_NODE
Specifies that the filters are specific to a node.

Note: Either the PERFFILT_ID or PERFFILT_NAME attribute values must be specified.

userbuff
Points to the memory area that is filled with the `perfstat_ssp_t` structure. Memory is allocated to this parameter with the calculation of `(sizeof (perfstat_ssp_t) * returned_count)`, where `returned_count` is the value obtained by setting this parameter to NULL and the `desired_number` parameter to zero.

sizeof_struct
Specifies the size of the `perfstat_ssp_t` structure.

desired_number
Specifies the number of the `perfstat_ssp_t` structures to copy to the **userbuff** parameter.

ssp_flag
Specifies whether tier, failure group, or physical volume needs to be retrieved. You must set this parameter to one of the following values:

SSPTIER
When the **SSPTIER** flag is invoked, the **userbuff** parameter is set to NULL, and the **desired_number** parameter is set to 0, the number of tiers based on the **name** parameter in the SSP is returned. When the **userbuff** parameter is allocated, tier-specific data is populated into the user buffer. The **name->spec** parameter can be used with the following specifications:

- `PERFFILT_ID` or `PERFFILT_NAME`
- `PERFFILT_TIER`

SSPFG
When the **SSPFG** flag is invoked, the **userbuff** parameter is set to NULL, and the **desired_number** parameter is set to 0, the number of failure groups based on the **name** parameter in the SSP is returned. When the **userbuff** parameter is allocated, the failure group-specific data is populated into the user buffer based on the name parameter. The **name.spec** flag can be used with the following specifications for the filter:

- `PERFFILT_ID` or `PERFFILT_NAME`
- `PERFFILT_TIER`
- `PERFFILT_FG`

SSPPV
When the **SSPPV** flag is invoked, the **userbuff** parameter is set to NULL, and the **desired_number** parameter is set to 0, the number of physical volumes based on the **name** parameter in the SSP is returned. When the **userbuff** parameter is allocated, the tier-specific data is populated into the user buffer. The **name.spec** flag can be used with the following specifications for the filter:

- `PERFFILT_ID` or `PERFFILT_NAME`
- `PERFFILT_TIER`
- `PERFFILT_FG`
- `PERFFILT_PV`

SSPNODE
When the **SSPNODE** flag is invoked, the **userbuff** parameter is set to NULL, and the **desired_number** parameter is set o 0, the number of nodes based on the name parameter in the SSP is returned. When the **userbuff** parameter is allocated, the node-specific data is populated into the user buffer. The **name.spec** flag can be used with the following specifications for the filter:

- `PERFFILT_NAME`
- `PERFFILT_NODE`

Return values

On successful completion of the subroutine, the number of filled structures is returned. If the subroutine is unsuccessful, a value of -1 is returned and the *errno* variable indicates the error.

Error codes

The **perfstat_ssp_ext** subroutine fails because of one of the following errors:

EINVAL
One of the parameters is not valid.

EFAULT
The memory is not sufficient.

ENOMEM
The default length of the string is short.

ENOMSG
The dictionary is not accessible.

ETIMEDOUT
The connection timed out.

ENOENT
Data specified by the filter is not found.

Files

The `libperfstat.h` file defines standard macros, data types, and subroutines.

perfstat_tape Subroutine

Purpose

Retrieves individual tape use statistics

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_tape (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_tape_t * userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_tape** subroutine retrieves one or more tape use statistics. It can also be used to retrieve the number of available sets of tape.

To get one or more sets of tape use metrics, specify the first tape for which statistics are to be collected in the *name* parameter, and set the *desired_number* parameter. To start from the first tape, specify the quotation marks (“”) or FIRST_TAPE as the name. The *userbuff* parameter must always point to the memory area big enough to contain the desired number of **perfstat_tape_t** structures that this subroutine is to copy. Upon return, the *name* parameter is set to either the name of the next tape, or to “” after all of the structures are copied.

To retrieve the number of available sets of tape use metrics, set the *name* parameter and the *userbuff* parameter to the value of null, and set the *desired_number* parameter to the value of zero. The returned value is the number of available sets.

Parameters

Item	Description
name	Contains the quotation marks (“”), FIRST_TAPE, or the name indicating the first tape for which the statistics are to be collected
userbuff	Points to the memory that is to be filled with the **perfstat_tape_t** structure
sizeof_struct	Specifies the size of the **perfstat_tape_t** structure
desired_number	Specifies the number of different tape statistics to be collected

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned.

Error Codes

The **perfstat_tape** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient
ENOMEM	The default length of the string is too short
ENOMSG	Cannot access dictionary

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_tape_total Subroutine

Purpose

Retrieves global tape use statistics

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_tape_total (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_tape_total_t * userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_tape_total** subroutine global tape use statistics in the **perfstat_tape_total_t** structure.

To get the statistics of tape use that are global to the whole system, the *name* parameter must be set to the value of null, the *userbuff* parameter must be allocated, and the value of the *desired_number* parameter must be set to the value of one.

This subroutine is not supported inside a WPAR.

Parameters

Item	Description
name	Contains the quotation marks (“”), FIRST_TAPE, or the name indicating the first tape for which the statistics are to be collected
userbuff	Points to the memory that is to be filled with the **perfstat_tape_t** structure
sizeof_struct	Specifies the size of the **perfstat_tape_t** structure
desired_number	Specifies the number of different tape statistics to be collected

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned.

Error Codes

The **perfstat_tape** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient
ENOMEM	The default length of the string is too short

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_thread Subroutine

Purpose

Retrieves kernel thread utilization metrics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_thread (name,userbuff,sizeof_userbuff,desired_number)
perfstat_id_t* name;
perfstat_thread_t* userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_thread** subroutine is used to retrieve per kernel thread utilization metrics for a process or for all the processes. The **perfstat_thread** subroutine retrieves one or more kernel thread statistics to populate the **perfstat_thread_t** data structure.

If the *name* and *userbuff* parameters are set as NULL, and the *desired_number* parameter is set to 0, the **perfstat_thread** subroutine returns the number of active threads.

If the *name* and *userbuff* parameters are set to NULL, and the *sizeof_userbuff* parameter is set to 0, the size of the current version of the **perfstat_thread_t** data structure is returned.

Parameters

Item	Description
name	Determines whether the kernel thread statistics must be captured for all the processes or captured for a specific process. The **name** parameter, must be set to NULL to get the kernel thread statistics for all processes. To get the kernel thread statistics for a specific process, the process ID must be specified. **Note:** The value of the ID must be passed as a string to the *name* parameter. For example, to retrieve the statistics for a process that has the process ID 12345, the *name* parameter must be set to 12345.
userbuff	Points to the memory area that is filled with one or more **perfstat_thread_t** data structures.
sizeof_userbuff	Specifies the size of the **perfstat_thread_t** data structure. **Note:** To obtain the size of the latest version of the **perfstat_thread_t** data structure, set the *sizeof_userbuff* parameter to 0, and the *name* and *userbuff* parameter to NULL.
desired_number	Specifies the number of **perfstat_thread_t** data structures to copy to the *userbuff* parameter.

Return Values

Unless the **perfstat_thread** subroutine is used to retrieve the number of available structures, the number of structures that are filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_thread_util Subroutine

Purpose

Calculates thread utilization metrics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_thread_util (data,userbuff,sizeof_userbuff,desired_number)
perfstat_rawdata_t* data;
perfstat_thread_t* userbuff;
int sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_thread_util** subroutine provides the interface for thread utilization metrics. The **perfstat_thread** subroutine retrieves one or more kernel thread statistics to populate the **perfstat_thread_t** data structure. The **perfstat_thread_util** subroutine uses the current and previous values to calculate the utilization metrics.

If the *name* and *userbuff* parameters are set to NULL and the *sizeof_userbuff* parameter is set to 0, the size of the current version of the **perfstat_thread_t** data structure is returned.

If the *desired_number* parameter is set to 0, the number of current elements from the **perfstat_rawdata_t** data structure is returned.

Parameters

Item	Description
data	Specifies that the data parameter is of the type **perfstat_rawdata_t**. The **perfstat_rawdata_t** data structure uses the current and the previous values to calculate the utilization metrics.
userbuff	Points to the memory area that is filled with one or more **perfstat_thread_t** data structures.
sizeof_userbuff	Specifies the size of the **perfstat_thread_t** data structure. **Note:** To obtain the size of the latest version of the **perfstat_thread_t** data structure, set the *sizeof_userbuff* parameter to 0, and the *name* and *userbuff* parameter to NULL.
desired_number	Specifies the number of **perfstat_thread_t** data structures to copy to the *userbuff* parameter.

Return Values

Unless the **perfstat_thread_util** subroutine is used to retrieve the number of available structures, the number of structures that are filled is returned upon successful completion. If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	One of the parameters is not valid.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_virtualdiskadapter Subroutine

Purpose

Retrieves the Virtual Small Computer System Interface (SCSI) or Serial Attached SCSI (SAS) adapter usage statistics in Virtual I/O Server (VIOS).

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_virtualdiskadapter (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_diskadapter_t * userbuff;
size_t sizeof_struct;int desired_number;
```

Description

The **perfstat_virtualdiskadapter** subroutine retrieves one or more Virtual SCSI/SAS adapter usage statistics.

The same function can also be used to retrieve the number of available sets of Virtual SCSI/SAS adapter (VHOST) statistics.

To get one or more sets of Virtual SCSI usage metrics, set the *name* parameter to the name of the first Virtual SCSI adapter for which the statistics are to be collected, and set the *desired_number* parameter. To start from the first Virtual SCSI adapter, set the *name* parameter to the quotation marks (" ") or *FIRST_VHOST*. The *userbuff* parameter must always point to the memory area that is big enough to contain the number of **perfstat_diskadapter_t** structures that this subroutine is to copy. Upon return, the *name* parameter is set to either the name of the next network adapter, to the quotation marks (" ") after all of the structures are copied.

To retrieve the number of available sets of Virtual SCSI adapter usage metrics, set the *name* parameter and the *userbuff* parameter to the value of null, and the *desired_number* parameter to the value of zero. The returned value is the number of available vhost adapter. The **perfstat_virtualdiskadapter** subroutine provides the statistics only in VIOS machine.

Parameters

Item	Description
name	Contains the quotation marks (" "), FIRST_VHOST, or the name indicating the first network adapter volume group for which the statistics is to be retrieved. For example: vhost0, vhost1.
userbuff	Points to the memory that is to be filled with one or more **perfstat_diskadapter_t** structures.
sizeof_struct	Specifies the size of the **perfstat_diskadapter_t** structure.
desired_number	Specifies the number of **perfstat_diskadapter_t** structures to copy to the *userbuff* parameter.

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned.

Error Codes

The **perfstat_virtualdiskadapter** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient
ENOMEM	The default length of the string is too short.
ENOMSG	The dictionary is not accessible.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_virtualdisktarget Subroutine

Purpose

Retrieves the Virtual Target Device (VTD) usage statistics in Virtual I/O Server (VIOS).

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_virtualdisktarget (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_disk_t * userbuff;
size_t sizeof_struct;int desired_number;
```

Description

The **perfstat_virtualdisktarget** subroutine retrieves one or more virtual target device usage statistics.

The same function can also be used to retrieve the number of available sets of virtual target device usage statistics.

To get one or more sets of virtual target device usage metrics, set the *name* parameter to the name of the first virtual target device for which the statistics are to be collected, and set the *desired_number* parameter. To start from the first virtual target device, set the *name* parameter to the quotation marks (" ") or *FIRST_VTD*. The *userbuff* parameter must always point to the memory area that is big enough to contain the number of **perfstat_disk_t** structures that this subroutine is to copy. Upon return, the *name* parameter is set to either the name of the next network adapter, or to the quotation marks (" ") after all of the structures are copied.

To retrieve the number of available sets of virtual target device usage metrics, set the *name* parameter and the *userbuff* parameter to the value of null, and the *desired_number* parameter to the value of zero. The returned value is the number of available sets. The **perfstat_virtualdisktarget** subroutine provides the statistics only in VIOS machine.

The following **perfstat_disk_t** structure fields are not filled by the *perfstat_virtualdisktarget* subroutine:

- *description*
- *vgname*
- *size*
- *free*

- *qdepth*
- *adapter*
- *paths_count*
- *wpar_id*

Parameters

Item	Description
name	Contains the quotation marks (" "), FIRST_VTD, or the name indicating the first network adapter for which the statistics is to be retrieved. For example: vtscsi0, vtscsi1.
userbuff	Points to the memory that is to be filled with one or more **perfstat_disk_t** structures.
sizeof_struct	Specifies the size of the **perfstat_disk_t** structure.
desired_number	Specifies the number of **perfstat_disk_t** structures to copy to the *userbuff* parameter.

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_virtualdisktarget** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient
ENOMEM	The default length of the string is too short.
ENOMSG	The dictionary is not accessible.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_virtual_fcadapter Subroutine

Purpose

Retrieves the Virtual Fiber Channel adapter (NPIV) usage statistics.

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_virtual_fcadapter (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_fcstat_t * userbuff;
size_t sizeof_struct;int desired_number;
```

Description

The **perfstat_virtual_fcadapter** subroutine retrieves one or more Virtual Fiber Channel adapter (NPIV) usage statistics.

The same function can also be used to retrieve the number of available sets of Virtual Fiber Channel adapter (NPIV) usage statistics.

To get one or more sets of Virtual FC adapter (NPIV) usage metrics, set the *name* parameter to the name of the first Virtual FC adapter for which the statistics are to be collected, and set the *desired_number* parameter. To start from the first Virtual FC adapter, set the *name* parameter to the quotation marks (" ") or *FIRST_VFCHOST*. The *userbuff* parameter must always point to the memory area that is big enough to contain the number of **perfstat_fcstat_t** structures that this subroutine is to copy. Upon return, the *name* parameter is set to either the name of the next FC adapter, or to the quotation marks (" ") after all of the structures are copied.

To retrieve the number of available sets of Virtual FC adapter usage metrics, set the *name* parameter and the *userbuff* parameter to the value of null, and the *desired_number* parameter to the value of zero. The returned value is the number of available sets.

Parameters

Item	Description
name	Contains the quotation marks (" "), FIRST_VFCHOST, or the name indicating the first FC adapter for which the statistics is to be retrieved. For example: vfchost0, vfchost1.
userbuff	Points to the memory that is to be filled with one or more **perfstat_fcstat_t** structures.
sizeof_struct	Specifies the size of the **perfstat_fcstat_t** structure.
desired_number	Specifies the number of **perfstat_fcstat_t** structures to copy to the *userbuff* parameter.

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned and the **errno** global variable is set.

Error Codes

The **perfstat_virtual_fcadapter** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient
ENOMEM	The default length of the string is too short.
ENOMSG	The dictionary is not accessible.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_volumegroup Subroutine

Purpose

Retrieves volume group related metrics

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>
int perfstat_volumegroup (name, userbuff, sizeof_struct, desired_number)
perfstat_id_t * name;
perfstat_volumegroup_t * userbuff;
int sizeof_userbuff;int desired_number;
```

Description

The **perfstat_volumegroup** subroutine retrieves one or more volume group statistics. It can also be used to retrieve the number of available volume group.

To get one or more sets of volume group metrics, set the *name* parameter to the name of the first volume group for which the statistics are to be collected, and set the *desired_number* parameter. To start from the first volume group, specify the quotation marks (“”) or FIRST_LOGICALVOLUME as the name. The *userbuff* parameter must always point to the memory area that is big enough to contain the number of **perfstat_volumegroup_t** structures that this subroutine is to copy. Upon return, the *name* parameter is set to either the name of the next volume group, or to “” after all of the structures are copied.

To retrieve the number of available sets of volume group metrics, set the *name* parameter and the *userbuff* parameter to the value of null, and the *desired_number* parameter to the value of zero. The returned value is the number of available volume groups.

Note: The **perfstat_config** must be called to enable the volume group statistics collection. The **perfstat_volumegroup** subroutine is not supported inside workload partitions.

Parameters

Item	Description
name	Contains the quotation marks (“”), FIRST_VOLUMEGROUP, or the name indicating the volume group for which the statistics is to be retrieved
userbuff	Points to the memory that is to be filled with the **perfstat_volumegroup_t** structure
sizeof_struct	Specifies the size of the **perfstat_volumegroup_t** structure
desired_number	Specifies the number of different volume group statistics to be collected

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned.

Error Codes

The **perfstat_volumegroup** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid
EFAULT	The memory is not sufficient

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perfstat_wpar_total Subroutine

Purpose

Retrieves workload partition (WPAR) use statistics

Library

Perfstat Library (**libperfstat.a**)

Syntax

```
#include <libperfstat.h>

int perfstat_wpar_total ( name, userbuff, sizeof_userbuff, desired_number )
perfstat_id_wpar_t *name;
perfstat_wpar_total_t *userbuff;
size_t sizeof_userbuff;
int desired_number;
```

Description

The **perfstat_wpar_total** subroutine returns the workload partition (WPAR) use statistics in the **perfstat_wpar_total_t** structure.

To get the total number of WPAR, the *name* parameter and the *userbuff* parameter must be specified as NULL, and the *desired_number* parameter must be specified as the value of zero.

To get the statistics of any particular WPAR, the WPAR ID or name must be specified in the *name* parameter. The *userbuff* parameter must be allocated. The *desired_number* parameter must be set. When this subroutine is called inside a WPAR, the *name* parameter must be set to NULL.

Parameters

Item	Description
name	Specifies the WPAR ID or the WPAR name. It is NULL if the subroutine is called from WPAR.
userbuff	Points to the memory area that is to be filled with the **perfstat_wpar_total_t** structure.
sizeof_userbuff	Specifies the size of the **perfstat_wpar_total_t** structure.
desired_number	Specifies the number of structures to return. The value of this parameter must be set to one.

Return Values

Upon successful completion, the number of structures filled is returned.

If unsuccessful, a value of -1 is returned, and the **errno** global variable is set.

Error Codes

The **perfstat_wpar_total** subroutine is unsuccessful if one of the following is true:

Item	Description
EINVAL	One of the parameters is not valid.
EFAULT	The memory is not sufficient.

Files

The **libperfstat.h** file defines standard macros, data types, and subroutines.

perror Subroutine

Purpose

Writes a message explaining a subroutine error.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <errno.h>
#include <stdio.h>
```

void perror (*String*)
const char **String*;

```
extern int errno;
extern char *sys_errlist[ ];
extern int sys_nerr;
```

Description

The **perror** subroutine writes a message on the standard error output that describes the last error encountered by a system call or library subroutine. The error message includes the *String* parameter string followed by a : (colon), a space character, the message, and a new-line character. The *String* parameter string should include the name of the program that caused the error. The error number is taken from the **errno** global variable, which is set when an error occurs but is not cleared when a successful call to the **perror** subroutine is made.

To simplify various message formats, an array of message strings is provided in the **sys_errlist** structure or use the **errno** global variable as an index into the **sys_errlist** structure to get the message string without the new-line character. The largest message number provided in the table is **sys_nerr**. Be sure to check the **sys_nerr** structure because new error codes can be added to the system before they are added to the table.

The **perror** subroutine retrieves an error message based on the language of the current locale.

After successfully completing, and before a call to the **exit** or **abort** subroutine or the completion of the **fflush** or **fclose** subroutine on the standard error stream, the **perror** subroutine marks for update the `st_ctime` and `st_mtime` fields of the file associated with the standard error stream.

Parameter

Item	Description
String	Specifies a parameter string that contains the name of the program that caused the error. The ensuing printed message contains this string, a : (colon), and an explanation of the error.

pipe Subroutine

Purpose

Creates an interprocess channel.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int pipe ( FileDescriptor)
int FileDescriptor[2];
```

Description

The **pipe** subroutine creates an interprocess channel called a pipe and returns two file descriptors, *FileDescriptor***[**0**]** and *FileDescriptor***[1]**. *FileDescriptor***[**0**]** is opened for reading and *FileDescriptor***[1]** is opened for writing.

A read operation on the *FileDescriptor***[**0**]** parameter accesses the data written to the *FileDescriptor***[1]** parameter on a first-in, first-out (FIFO) basis.

Write requests of **PIPE_BUF** bytes or fewer will not be interleaved (mixed) with data from other processes doing writes on the same pipe. **PIPE_BUF** is a system variable described in the **pathconf** subroutine. Writes of greater than **PIPE_BUF** bytes may have data interleaved, on arbitrary boundaries, with other writes.

If **O_NONBLOCK** or **O_NDELAY** are set, writes requests of **PIPE_BUF** bytes or fewer will either succeed completely or fail and return -1 with the **errno** global variable set to **EAGAIN**. A write request for more than **PIPE_BUF** bytes will either transfer what it can and return the number of bytes actually written, or transfer no data and return -1 with the **errno** global variable set to **EAGAIN**.

Parameters

Item	Description
FileDescriptor	Specifies the address of an array of two integers into which the new file descriptors are placed.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and the **errno** global variable is set to identify the error.

Error Codes

The **pipe** subroutine is unsuccessful if one or more the following are true:

Item	Description
EFAULT	The *FileDescriptor* parameter points to a location outside of the allocated address space of the process.
EMFILE	The number of open of file descriptors exceeds the **OPEN_MAX** value.
ENFILE	The system file table is full, or the device containing pipes has no free i-nodes.

plock Subroutine

Purpose

Locks the process, text, or data in memory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/lock.h>
```

```
int plock ( Operation)
int Operation;
```

Description

The **plock** subroutine allows the calling process to lock or unlock its text region (text lock), its data region (data lock), or both its text and data regions (process lock) into memory. The **plock** subroutine does not lock the shared text segment or any shared data segments. Locked segments are pinned in memory and are immune to all routine paging. Memory locked by a parent process is not inherited by the children after a **fork** subroutine call. Likewise, locked memory is unlocked if a process executes one of the **exec** subroutines. The calling process must have the root user authority to use this subroutine.

A real-time process can use this subroutine to ensure that its code, data, and stack are always resident in memory.

Note: Before calling the **plock** subroutine, the user application must lower the maximum stack limit value using the **ulimit** subroutine.

Parameters

Item	Description
Operation	Specifies one of the following: **PROCLOCK** Locks text and data into memory (process lock). **TXTLOCK** Locks text into memory (text lock). **DATLOCK** Locks data into memory (data lock). **UNLOCK** Removes locks.

Return Values

Upon successful completion, a value of 0 is returned to the calling process. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **plock** subroutine is unsuccessful if one or more of the following is true:

Item	Description
EPERM	The effective user ID of the calling process does not have the root user authority.
EINVAL	The *Operation* parameter has a value other than **PROCLOCK**, **TXTLOCK**, **DATLOCK**, or **UNLOCK**.
EINVAL	The *Operation* parameter is equal to **PROCLOCK**, and a process lock, text lock, or data lock already exists on the calling process.
EINVAL	The *Operation* parameter is equal to **TXTLOCK**, and a text lock or process lock already exists on the calling process.
EINVAL	The *Operation* parameter is equal to **DATLOCK**, and a data lock or process lock already exists on the calling process.
EINVAL	The *Operation* parameter is equal to **UNLOCK**, and no type of lock exists on the calling process.

pm_clear_ebb_handler Subroutine

Purpose

Clears the Event-Based Branching (EBB) facility configured for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int pm_clear_ebb_handler (void ** old_handler, void ** old_data_area)
```

Description

The **pm_clear_ebb_handler** subroutine clears the EBB facility that is previously configured for the calling thread, through the **pm_set_ebb_handler** subroutine.

Note: The **pm_clear_ebb_handler** subroutine can only be called when the thread mode is 1:1 and when counting for the thread is not started.

Parameters

Item	Description
old_handler	The old EBB handler configured for the thread. The value can be set to NULL if it is not required.
old_data_area	The old EBB data area. The value can be set to NULL if it is not required.

Return Values

If unsuccessful, a value other than zero is returned and a positive error code is set. If successful, a value of zero is returned.

Error Codes

The subroutine is unsuccessful if one of the following error codes are returned:

Item	Description
Pmapi_NoInit	The **pm_initialize** subroutine is not called.
Pmapi_Unsupported_EBBThreadMode	The thread is not running in the 1:1 mode.
Pmapi_NoSetProg	The **pm_set_program** subroutine is not called.
Pmapi_Invalid_EBB_Config	The *PTHREAD_EBB_PMU_TYPE* flag is not passed to the **pthread** subroutine.
Pmapi_EBB_NotSet	The EBB handler is not set by the caller.
Non-zero error codes	Returned by the **pthread** call or the **pmsvcs** call.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

pm_cycles Subroutine

Purpose

Returns processor speed in cycles per second.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

double pm_cycles (void)
```

Description

The **pm_cycles** subroutine returns the *nominal processor speed* for the system. The speed is returned in cycles per second. The nominal processor speed is the maximum frequency at which the system can run across all environments and workload conditions. Depending on system conditions, the nominal processor frequency might not represent the minimum or maximum achievable processor speed.

Return Values

Item	Description
0	An error occurred.
Processor speed in cycles per second	No errors occurred.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_delete_program and pm_delete_program_wp Subroutines

Purpose

Deletes previously established system-wide Performance Monitor settings.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_delete_program ()
int pm_delete_program_wp (cid_t cid)
```

Description

The **pm_delete_program** subroutine deletes previously established system-wide Performance Monitor settings.

The **pm_delete_program_wp** subroutine deletes previously established system-wide Performance Monitor settings for a specified workload partition (WPAR).

Parameters

Item	Description
cid	Specifies the identifier of the WPAR for which the programming is to be deleted. The CID can be obtained from the WPAR name using the **getcorralid** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_delete_program_group Subroutine

Purpose

Deletes previously established Performance Monitor settings for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_delete_program_group ( pid,  tid)
pid_t pid;
tid_t tid;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_delete_program_pgroup** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_delete_program_pgroup** subroutine with a *ptid* parameter equal to 0.

The **pm_delete_program_group** subroutine deletes previously established Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process. The settings for the group to which the target thread belongs and from all the other threads in the same group are also deleted.

Parameters

Item	Description
pid	Process identifier of target thread. The target process must be a debuggee under the control of the calling process.
tid	Thread identifier of a target thread.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the pm_error (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the pm_error (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_delete_program_mygroup Subroutine

Purpose

Deletes previously established Performance Monitor settings for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_delete_program_mygroup ()
```

Description

The **pm_delete_program_mygroup** subroutine deletes previously established Performance Monitor settings for the calling kernel thread, the counting group to which it belongs, and for all the threads that are members of the same group.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_delete_program_mythread Subroutine

Purpose

Deletes the previously established Performance Monitor settings for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_delete_program_mythread ()
```

Description

The **pm_delete_program_mythread** subroutine deletes the previously established Performance Monitor settings for the calling kernel thread.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_delete_program_pgroup Subroutine

Purpose

Deletes previously established Performance Monitor settings for the counting group to which a target pthread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_delete_program_pgroup ( pid,  tid,  ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;
```

Description

The **pm_delete_program_pgroup** subroutine deletes previously established Performance Monitor settings for a target pthread. The pthread must be stopped and must be part of a debuggee process under the control of the calling process. The settings for the group to which the target pthread belongs and from all the other pthreads in the same group are also deleted.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

Parameters

Item	Description
pid	Process ID of target thread. The target process must be a debuggee under the control of the calling process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the"pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_delete_program_pthread Subroutine

Purpose

Deletes the previously established Performance Monitor settings for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_delete_program_pthread ( pid,  tid,  ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;
```

Description

The **pm_delete_program_pthread** subroutine deletes the previously established Performance Monitor settings for a target pthread. The pthread must be stopped and must be part of a debuggee process under the control of the calling process.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee under the control of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the“pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_delete_program_thread Subroutine

Purpose

Deletes the previously established Performance Monitor settings for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_delete_program_thread ( pid,  tid)
pid_t pid;
tid_t tid;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_delete_program_pthread** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_delete_program_pthread** subroutine with a *ptid* parameter equal to 0.

The **pm_delete_program_thread** subroutine deletes the previously established Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process.

Parameters

Item	Description
pid	Process identifier of target thread. Target process must be a debuggee under the control of the calling process.
tid	Thread identifier of the target thread.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_disable_bhrb Subroutine

Purpose

Disables all Branch History Rolling Buffer (BHRB) related instructions such as **clrbhrb** and **mfbhrb** in problem state.

Note: The **pm_disable_bhrb** subroutine can only be called when the thread mode is 1:1 and when counting for the thread is not started.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int pm_disable_bhrb (void)
```

Description

The **pm_disable_bhrb** subroutine disables the BHRB instructions like **clrbhrb** and **mfbhrb** in problem state.

If the BHRB instructions are disabled in the problem state, the Facility Unavailable interrupt is generated when these instructions are used in the problem state.

Return Values

If unsuccessful, a value other than zero is returned and a positive error code is set. If successful, a value of zero is returned.

Error Codes

The subroutine is unsuccessful if the following is true:

Item	Description
Pmapi_NoInit	The **pm_initialize** subroutine is not called.
Pmapi_NoSetProg	The **pm_set_program** subroutine is not called.
Other non-zero error codes	Returned by the **pmsvcs** subroutine.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

pm_enable_bhrb Subroutine

Purpose

Enables all Branch History Rolling Buffer (BHRB) related instructions such as **clrbhrb** and **mfbhrb** in the problem state and configures the Branch History Rolling Buffer Enable (BHRBE) filtering modes.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int pm_enable_bhrb (pm_bhrb_ifm_t ifm_mode)
```

Description

The **pm_enable_bhrb** subroutine enables the BHRB instructions such as **clrbhrb** and **mfbhrb** in the problem state and configures the BHRBE filtering modes.

Note: The **pm_enable_bhrb** subroutine can only be called when the thread mode is 1:1 and when counting for the thread is not started.

Parameters

Item	Description
ifm_mode	BHRBE filtering mode.

The *ifm_mode* parameter can take one of the following values as defined in the **pm_bhrb_ifm_t** structure:

```
typedef enum
{
    BHRB_IFM0 = 0,
    BHRB_IFM1,
    BHRB_IFM2,
```

```
    BHRB_IFM3
}pm_bhrb_ifm_t;
```

where,

- **BHRB_IFM0** - No filtering.
- **BHRB_IFM1** - Do not record any branch instructions unless the value of the *LK* field is set to 1.
- **BHRB_IFM2** - Do not record I-Form instructions. For the B-Form and XL-Form instructions for which the BO field indicates **Branch always**, do not record the instruction. If it is a B-Form instruction, do not record the instruction address but record only the branch target address. If it is a XL-Form, do not record the I-Form instructions.
- **BHRB_IFM3** - Filter and enter BHRB entries for the mode 10. For B-Form and XL-Form instructions for which the **BO** field is set to 1 or for which the **a** bit in the **BO** field is set to 1, do not record the instruction. If it is B-Form and do not record the instruction address but record only the branch target address if it is XL-Form.

When the BHRB is written by the hardware, only the Branch instructions that meet the filtering criteria and for which the branch are included are termed as BHRB entries (BHRBE).

Return Values

If unsuccessful, a value other than zero is returned and positive error code is set. If successful, a value of zero is returned.

Error Codes

The subroutine is unsuccessful if the following error codes are returned:

Item	Description
Pmapi_NoInit	The **pm_initialize** subroutine is not called.
Pmapi_NoSetProg	The **pm_set_program** subroutine is not called.
Pmapi_Invalid_IFMMode	The value of an *ifm_mode* is not valid.
Other non-zero error codes	Returned by the **pmsvcs** subroutine.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

pm_error Subroutine

Purpose

Decodes Performance Monitor APIs error codes.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

void pm_error ( *Where,  errorcode)
```

```
char *Where;
int errorcode;
```

Description

The **pm_error** subroutine writes a message on the standard error output that describes the parameter *errorcode* encountered by a Performance Monitor API library subroutine. The error message includes the *Where* parameter string followed by a : (colon), a space character, the message, and a new-line character. The *Where* parameter string includes the name of the program that caused the error.

Parameters

Item	Description
**Where*	Specifies where the error was encountered.
errorcode	Specifies the error code as returned by one of the Performance Monitor APIs library subroutines.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_generic subroutine

Purpose

Returns performance monitor data for the following threads and groups:

- Target thread.
- Target POSIX thread (pthread).
- The counting group of the target thread.
- The counter multiplexing mode for the target thread.
- The counter multiplexing mode for the counting group to which a target thread belongs.
- The counter multiplexing mode for a target pthread.

Library

Performance monitor APIs library (`libpmapi.a`)

Syntax

```
#include <pmapi.h>
int pm_get_data_generic (pid,tid,ptid, type,*pmdata)
pid_t pid;
tid_t tid;
ptid_t ptid;
profiler_type_t type;
pm_data_time_t *pmdata;
```

Description

The `pm_get_data_generic` subroutine retrieves the current performance monitor data based on parameters that are provided to the subroutine. If the pthread is running in 1:1 mode, only the **tid** parameter must be specified. If the pthread is running in m:n mode, only the **ptid** parameter must be specified.

If both the **ptid** and **tid** parameters are specified, the following conditions must be met:

- Both the **ptid** and **tid** parameters must refer to a single pthread.
- The thread must run on a kernel thread context with the specified **tid** parameter.

The performance monitor data is always a set of 64-bit values per hardware counter on the used system.

Parameters

pid
Process identifier of a target thread. The target thread must be a debuggee process of the caller process.

tid
Thread identifier of a target thread. You can assign a value of 0 to ignore this parameter.

ptid
pthread ID of the target pthread. You can assign a value of 0 to ignore this parameter.

type
Type of the target. The following are two types of targets:

P_THREAD
This flag is set if the target thread is a pthread.

P_THREAD_GROUP
This flag is set if the target is a group.

***pmdata**
Pointer to a structure to return the performance monitor data. The structure contains array of accumulated counters, accumulated time, accumulated Processor Utilization Resource Register (PURR) and Scalable Processor Utilization Resource register (SPURR) time for each event set that is counted for the target kernel thread.

Return values

The `pm_get_data_generic` subroutine returns 0 if no errors occurred during the subroutine execution and returns a positive error code otherwise. Use the `pm_error` subroutine to decode the error code.

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data, pm_get_tdata, pm_get_Tdata, pm_get_data_cpu, pm_get_tdata_cpu, pm_get_Tdata_cpu, pm_get_data_lcpu, pm_get_tdata_lcpu and pm_get_Tdata_lcpu Subroutine

Purpose

Returns systemwide Performance Monitor data.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data ( *pmdata)
pm_data_t *pmdata;
```

```
int pm_get_tdata (pmdata, * time)
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata (pmdata, * times)
pm_data_t *pmdata;
pm_accu_time_t *times;

int pm_get_data_cpu (cpuid, *pmdata)
int cpuid;
pm_data_t *pmdata;

int pm_get_tdata_cpu (cpuid, *pmdata, *time)
int cpuid;
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_cpu (cpuid, *pmdata, *times)
int cpuid;
pm_data_t *pmdata;
pm_accu_time_t *times
int pm_get_data_lcpu (lcpuid, *pmdata)
int lcpuid;
pm_data_t *pmdata;

int pm_get_tdata_lcpu (lcpuid, *pmdata, *time)
int lcpuid;
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_lcpu (lcpuid, *pmdata, *times)
int lcpuid;
pm_data_t *pmdata;
pm_accu_time_t *times
```

Description

The **pm_get_data** subroutine retrieves the current systemwide Performance Monitor data.

The **pm_get_tdata** subroutine retrieves the current systemwide Performance Monitor data, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata** subroutine retrieves the current systemwide Performance Monitor data, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The `pm_get_data_cpu`, `pm_get_tdata_cpu`, and `pm_get_Tdata_cpu` subroutines retrieve the current Performance Monitor data for a specified processor. The given processor ID represents a contiguous number ranging from `0` to `_system_configuration.ncpus`. These subroutines can only be used when no Dynamic Reconfiguration operations are made on the machine, because when processors are added or removed, the processor numbering is modified and the specified processor number can designate different processors from one call to another. These subroutines are maintained for compatibility with previous versions.

The **pm_get_data_cpu** subroutine retrieves the current Performance Monitor data for the specified processor.

The **pm_get_tdata_cpu** subroutine retrieves the current Performance Monitor data for the specified processor, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata_cpu** subroutine retrieves the current Performance Monitor data for the specified processor, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The `pm_get_data_lcpu`, `pm_get_tdata_lcpu`, and `pm_get_Tdata_lcpu` subroutines retrieve the current Performance Monitor data for a specified logical processor. The given processor ID represents a value ranging from `0` to `_system_configuration.max_ncpus`. This value always represents the same processor, even after Dynamic Reconfiguration operations have occurred. These subroutines might return an error if the specified logical processor number has never run during the counting interval.

The **pm_get_data_lcpu** subroutine retrieves the current Performance Monitor data for the specified logical processor.

The **pm_get_tdata_lcpu** subroutine retrieves the current Performance Monitor data for the specified logical processor, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata_lcpu** subroutine retrieves the current Performance Monitor data for the specified logical processor, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machines used) of 64-bit values.

Parameters

Item	Description
**pmdata*	Pointer to a structure that contains the returned systemwide Performance Monitor data.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
**times*	Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the **time_base_to_time** subroutine.
cpuid	Contiguous processor numbers ranging from `0` to `_system_configuration.ncpus`. This value does not always designate the same processor, even after Dynamic Reconfiguration operations have occurred.
lcpuid	Logical processor identifier. Each identifier stays linked to a particular processor between reboots, even after Dynamic Reconfiguration operations. This value must be in the range from `0` to `_system_configuartion.max_ncpus`.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Refer to the pm_error Subroutine to decode the error code.

Error Codes

Refer to the pm_error Subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_group, pm_get_tdata_group and pm_get_Tdata_group Subroutine

Purpose

Returns Performance Monitor data for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_group (pid,  tid,  *pmdata)
pid_t pid;
tid_t tid;
pm_data_t *pmdata;

int pm_get_tdata_group (pid, tid, *pmdata, *time)
pm_data_t *pmdata;
pid_t pid;
tid_t tid;
timebasestruct_t *time;

int pm_get_Tdata_group (pid, tid, *pmdata, *times)
pm_data_t *pmdata;
pid_t pid;
tid_t tid;
pm_accu_time_t *times;
```

Description

These subroutines support only the 1:1 threading model. They have been superseded by the **pm_get_data_pgroup** and **pm_get_tdata_pgroup** subroutines, which support both the 1:1 and the M:N threading models. Calls to these subroutines are equivalent to calls to the **pm_get_data_pgroup** and **pm_get_tdata_pgroup** subroutines with a *ptid* parameter equal to 0.

The **pm_get_data_group** subroutine retrieves the current Performance Monitor data for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process.

The **pm_get_tdata_group** subroutine retrieves the current Performance Monitor data for the counting group to which a target thread belongs, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata_group** subroutine retrieves the current Performance Monitor data for the counting group to which a target thread belongs, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the threads in the group.

Parameters

Item	Description
pid	Process identifier of a target thread. The target process must be an argument of a debug process.
tid	Thread identifier of a target thread.
**pmdata*	Pointer to a structure to return the Performance Monitor data for the group to which the target thread belongs.

Item	Description
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
**times*	Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_group_mx and pm_get_tdata_group_mx Subroutine

Purpose

Returns Performance Monitor data in counter multiplexing mode for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_group_mx (pid,  tid,  *pmdata)
pid_t pid;
tid_t tid;
pm_data_mx_t *pmdata;

int pm_get_tdata_group_mx (pid, tid, *pmdata, *time)
pm_data_mx_t *pmdata;
pid_t pid;
tid_t tid;
timebasestruct_t *time;
```

Description

These subroutines support only the 1:1 threading model. They have been superseded by the **pm_get_data_pgroup_mx** and **pm_get_tdata_pgroup_mx** subroutines, which support both the 1:1 and the M:N threading models. Calls to these subroutines are equivalent to calls to the **pm_get_data_pgroup_mx** and **pm_get_tdata_pgroup_mx** subroutines with a *ptid* parameter equal to 0.

The **pm_get_data_group_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process.

The **pm_get_tdata_group_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the counting group to which a target thread belongs, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, whether it is a process level group, and whether its counters are consistent with the sum of the counters for all of the threads in the group.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the *pmdata* parameter).

Parameters

Item	Description
pid	Process identifier of a target thread. The target process must be an argument of a debug process.
tid	Thread identifier of a target thread.
**pmdata*	Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the group to which the target thread belongs.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the“pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_mx, pm_get_tdata_mx, pm_get_data_cpu_mx, pm_get_tdata_cpu_mx, pm_get_data_lcpu_mx and pm_get_tdata_lcpu_mx Subroutine

Purpose

Returns systemwide Performance Monitor data in counter multiplexing mode.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_mx ( *pmdata)
pm_data_mx_t *pmdata;

int pm_get_tdata_mx (pmdata, * time)
pm_data_mx_t *pmdata;
timebasestruct_t *time;

int pm_get_data_cpu_mx (cpuid, *pmdata)
int cpuid;
pm_data_mx_t *pmdata;

int pm_get_tdata_cpu_mx (cpuid, *pmdata, *time)
int cpuid;
pm_data_mx_t *pmdata;
timebasestruct_t *time;

int pm_get_data_lcpu_mx (lcpuid, *pmdata)
int lcpuid;
pm_data_mx_t *pmdata;

int pm_get_tdata_lcpu_mx (lcpuid, *pmdata, *time)
int lcpuid;
pm_data_mx_t *pmdata;
timebasestruct_t *time;
```

Description

The **pm_get_data_mx** subroutine retrieves the current systemwide Performance Monitor data in counter multiplexing mode.

The **pm_get_tdata_mx** subroutine retrieves the current systemwide Performance Monitor data in counter multiplexing mode, and a timestamp indicating the last time the hardware counters were read.

The `pm_get_data_cpu_mx` and the `pm_get_tdata_cpu_mx` subroutines retrieve the current Performance Monitor data for a specified processor. The given processor ID represents a contiguous number ranging from `0` to `_system_configuration.ncpus`. These subroutines can only be used when no Dynamic Reconfiguration operations are made on the machine, because when processors are added or removed, the processor numbering is modified and the specified processor number can designate different processors from one call to another. These subroutines are maintained for compatibility with previous versions.

The **pm_get_data_cpu_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the specified processor.

The **pm_get_tdata_cpu_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the specified processor, and a timestamp indicating the last time the hardware counters were read.

The pm_get_data_lcpu_mx and the pm_get_tdata_lcpu_mx subroutines retrieve the current Performance Monitor data for a specified logical processor. The given processor ID represents a value ranging from 0 to _system_configuration.max_ncpus. This value always represents the same processor, even after Dynamic Reconfiguration operations have occurred. These subroutines might return an error if the specified logical processor number has never run during the counting interval.

The **pm_get_data_lcpu_mx** subroutine retrieves the current Performance Monitor data for the specified logical processor in counter multiplexing mode.

The **pm_get_tdata_lcpu_mx** subroutine retrieves the current Performance Monitor data for the specified logical processor in counter multiplexing mode, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of a set (one per hardware counter on the machines used) of 64-bit values.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the *pmdata* parameter).

Parameters

Item	Description
**pmdata*	Pointer to a structure that contains the returned systemwide Performance Monitor data. (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted)
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
cpuid	Contiguous processor numbers going from 0 to _system_configuration.ncpus. This value does not always designate the same processor, even after Dynamic Reconfiguration operations have occurred.
lcpuid	Logical processor identifier. Each identifier stays linked to a particular processor between reboots, even after Dynamic Reconfiguration operations. This value must be in the range from 0 to _system_configuartion.max_ncpus.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Refer to the pm_error Subroutine to decode the error code.

Error Codes

Refer to the pm_error Subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_mygroup, pm_get_tdata_mygroup or pm_get_Tdata_mygroup Subroutine

Purpose

Returns Performance Monitor data for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_mygroup (*pmdata)
pm_data_t *pmdata;

int pm_get_tdata_mygroup (*pmdata, *time)
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_mygroup (pmdata, * times)
pm_data_t *pmdata;
pm_accu_time_t *times;
```

Description

The **pm_get_data_mygroup** subroutine retrieves the current Performance Monitor data for the group to which the calling kernel thread belongs.

The **pm_get_tdata_mygroup** subroutine retrieves the current Performance Monitor data for the group to which the calling thread belongs, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata_mygroup** subroutine retrieves the current Performance Monitor data for the group to which the calling thread belongs, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the threads in the group.

Parameters

Item	Description
**pmdata*	Pointer to a structure to return the Performance Monitor data for the group to which the calling thread belongs.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
**times*	Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the `time_base_to_time` subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_mygroup_mx or pm_get_tdata_mygroup_mx Subroutine

Purpose

Returns Performance Monitor data in counter multiplexing mode for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_mygroup_mx (*pmdata)
pm_data_mx_t *pmdata;

int pm_get_tdata_mygroup_mx (*pmdata, *time)
pm_data_mx_t *pmdata;
timebasestruct_t *time;
```

Description

The **pm_get_data_mygroup_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the group to which the calling kernel thread belongs.

The **pm_get_tdata_mygroup_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the group to which the calling thread belongs, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the threads in the group.

The user application must free the array allocated to store accumulated counts and times (accu_set field of pmdata).

Parameters

Item	Description
**pmdata*	Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the group to which the calling thread belongs.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_mythread, pm_get_tdata_mythread or pm_get_Tdata_mythread Subroutine

Purpose

Returns Performance Monitor data for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_mythread (*pmdata)
pm_data_t *pmdata;

int pm_get_tdata_mythread (*pmdata, *time)
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_mythread (pmdata, * times)
pm_data_t *pmdata;
pm_accu_time_t *times;
```

Description

The **pm_get_data_mythread** subroutine retrieves the current Performance Monitor data for the calling kernel thread.

The **pm_get_tdata_mythread** subroutine retrieves the current Performance Monitor data for the calling kernel thread, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata_mythread** subroutine retrieves the current Performance Monitor data for the calling kernel thread, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values.

Parameters

Item	Description
**pmdata*	Pointer to a structure to contain the returned Performance Monitor data for the calling kernel thread.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
**times*	Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using `the time_base_to_time` subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_mythread_mx or pm_get_tdata_mythread_mx Subroutine

Purpose

Returns Performance Monitor data in counter multiplexing mode for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_mythread_mx (*pmdata)
pm_data_mx_t *pmdata;

int pm_get_tdata_mythread_mx (*pmdata, *time)
pm_data_mx_t *pmdata;
timebasestruct_t *time;
```

Description

The **pm_get_data_mythread_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the calling kernel thread.

The **pm_get_tdata_mythread_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the calling kernel thread, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the *pmdata* parameter).

Parameters

Item	Description
**pmdata*	Pointer to a structure to contain the returned Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the calling kernel thread.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_pgroup, pm_get_tdata_pgroup and pm_get_Tdata_pgroup Subroutine

Purpose

Returns Performance Monitor data for the counting group to which a target pthread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_pgroup (pid,  tid,  ptid,  *pmdata)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_t *pmdata;

int pm_get_tdata_pgroup (pid, tid, *pmdata, *time)
pm_data_t *pmdata;
pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time;

int pm_get_Tdata_pgroup (pid, tid, *pmdata, * times)
pm_data_t *pmdata;
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_accu_time_t *times;
```

Description

The **pm_get_data_pgroup** subroutine retrieves the current Performance Monitor data for the counting group to which a target pthread belongs. The pthread must be stopped and must be part of a debuggee process under the control of the calling process.

The **pm_get_tdata_pgroup** subroutine retrieves the current Performance Monitor data for the counting group to which a target pthread belongs, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata_pgroup** subroutine retrieves the current Performance Monitor data for the counting group to which a target pthread belongs, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, if it is a process level group, and if its counters are consistent with the sum of the counters for all of the pthreads in the group.

Parameters

Item	Description
pid	Process identifier of a target thread. The target process must be an argument of a debug process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**pmdata*	Pointer to a structure to return the Performance Monitor data for the group to which the target pthread belongs.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
**times*	Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the `time_base_to_time` subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_pgroup_mx and pm_get_tdata_pgroup_mx Subroutine

Purpose

Returns Performance Monitor data in counter multiplexing mode for the counting group to which a target pthread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_pgroup_mx (pid,  tid,  ptid,  *pmdata)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_mx_t *pmdata;

int pm_get_tdata_pgroup_mx (pid, tid, *pmdata, *time)
pm_data_mx_t *pmdata;
pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time;
```

Description

The **pm_get_data_pgroup_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the counting group to which a target pthread belongs. The pthread must be stopped and must be part of a debuggee process under the control of the calling process.

The **pm_get_tdata_pgroup_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for the counting group to which a target pthread belongs, and a timestamp indicating the last time the hardware counters were read.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values. The information returned also includes the characteristics of the group, such as the number of its members, whether it is a process level group, and whether its counters are consistent with the sum of the counters for all of the pthreads in the group.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the *pmdata* parameter).

Parameters

Item	Description
pid	Process identifier of a target thread. The target process must be an argument of a debug process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**pmdata*	Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the group to which the target pthread belongs.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_pthread, pm_get_tdata_pthread or pm_get_Tdata_pthread Subroutine

Purpose

Returns Performance Monitor data for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_pthread (pid,  tid,  ptid,  *pmdata)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_t *pmdata;

int pm_get_tdata_pthread (pid, tid, ptid, *pmdata, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_pthread (pid, tid, ptid,*pmdata, * times)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_t *pmdata;
pm_accu_time_t *times;
```

Description

The **pm_get_data_pthread** subroutine retrieves the current Performance Monitor data for a target pthread. The pthread must be stopped and must be part of a debuggee process under the control of a calling process.

The **pm_get_tdata_pthread** subroutine retrieves the current Performance Monitor data for a target pthread, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata_pthread** subroutine retrieves the current Performance Monitor data for a target pthread, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**pmdata*	Pointer to a structure to return the Performance Monitor data for the target pthread.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
**times*	Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_pthread_mx or pm_get_tdata_pthread_mx Subroutine

Purpose

Returns Performance Monitor data in counter multiplexing mode for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_pthread_mx (pid, tid, ptid, *pmdata)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_mx_t *pmdata;

int pm_get_tdata_pthread_mx (pid, tid, ptid, *pmdata, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_data_mx_t *pmdata;
timebasestruct_t *time;
```

Description

The **pm_get_data_pthread_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for a target pthread. The pthread must be stopped and must be part of a debuggee process under the control of a calling process.

The **pm_get_tdata_pthread_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for a target pthread, and a timestamp indicating the last time the hardware counters were read.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the *pmdata* parameter).

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Item	Description
**pmdata*	Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the target pthread.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_thread, pm_get_tdata_thread or pm_get_Tdata_thread Subroutine

Purpose

Returns Performance Monitor data for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_thread (pid,  tid,  *pmdata)
pid_t pid;
tid_t tid;
pm_data_t *pmdata;

int pm_get_tdata_thread (pid, tid, *pmdata, *time)
pid_t pid;
tid_t tid;
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_thread (pid, tid, *pmdata, * times)
pm_data_t *pmdata;
pid_t pid;
tid_t tid;
```

```
pm_data_t *pmdata;
pm_accu_time_t *times;
```

Description

These subroutines support only the 1:1 threading model. They have been superseded by the **pm_get_data_pthread** and **pm_get_tdata_pthread** subroutines, which support both the 1:1 and the M:N threading models. Calls to these subroutines are equivalent to calls to the **pm_get_data_pthread** and **pm_get_tdata_pthread** subroutines with a *ptid* parameter equal to 0.

The **pm_get_data_thread** subroutine retrieves the current Performance Monitor data for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of a calling process.

The **pm_get_tdata_thread** subroutine retrieves the current Performance Monitor data for a target thread, and a timestamp indicating the last time the hardware counters were read.

The **pm_get_Tdata_thread** subroutine retrieves the current Performance Monitor data for a target thread, and the accumulated time (timebase, PURR time and SPURR time) the events were counted.

The Performance Monitor data is always a set (one per hardware counter on the machine used) of 64-bit values.

Parameters

Item	Description
pid	Process identifier of a target thread. The target process must be a debuggee of the caller process.
tid	Thread identifier of a target thread.
**pmdata*	Pointer to a structure to return the Performance Monitor data for the target kernel thread.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
**times*	Pointer to a structure containing the accumulated time (timebase, PURR time and SPURR time) the events were counted. Each time counter can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_thread_mx or pm_get_tdata_thread_mx Subroutine

Purpose

Returns Performance Monitor data in counter multiplexing mode for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_thread_mx (pid,  tid,  *pmdata)
pid_t pid;
tid_t tid;
pm_data_mx_t *pmdata;

int pm_get_tdata_thread_mx (pid, tid, *pmdata, *time)
pid_t pid;
tid_t tid;
pm_data_mx_t *pmdata;
timebasestruct_t *time;
```

Description

These subroutines support only the 1:1 threading model. They have been superseded by the **pm_get_data_pthread_mx** and **pm_get_tdata_pthread_mx** subroutines, which support both the 1:1 and the M:N threading models. Calls to these subroutines are equivalent to calls to the **pm_get_data_pthread_mx** and **pm_get_tdata_pthread_mx** subroutines with a *ptid* parameter equal to 0.

The **pm_get_data_thread_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of a calling process.

The **pm_get_tdata_thread_mx** subroutine retrieves the current Performance Monitor data in counter multiplexing mode for a target thread, and a timestamp indicating the last time the hardware counters were read.

The Performance Monitor data is always an array of a set (one per hardware counter on the machine used) of 64-bit values.

The user application must free the array allocated to store accumulated counts and times (the accu_set field of the *pmdata* parameter).

Parameters

Item	Description
pid	Process identifier of a target thread. The target process must be a debuggee of the caller process.
tid	Thread identifier of a target thread.

Item	Description
**pmdata*	Pointer to a structure to return the Performance Monitor data (array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted) for the target kernel thread.
**time*	Pointer to a structure containing the timebase value the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_wp, pm_get_tdata_wp, pm_get_Tdata_wp, pm_get_data_lcpu_wp, pm_get_tdata_lcpu_wp, and pm_get_Tdata_lcpu_wp Subroutines

Purpose

Returns Performance Monitor data for a specified workload partition.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_wp (wp_handle, *pmdata)
pm_wp_handle_t wp_handle;
pm_data_t *pmdata;

int pm_get_tdata_wp (wp_handle, *pmdata, *time)
pm_wp_handle_t wp_handle;
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_wp (wp_handle, pmdata, * times)
pm_wp_handle_t wp_handle;
pm_data_t *pmdata;
pm_accu_time_t *times;
```

```
int pm_get_data_lcpu_wp (wp_handle, lcpuid, *pmdata)
pm_wp_handle_t wp_handle;
int lcpuid;
pm_data_t *pmdata;

int pm_get_tdata_lcpu_wp (wp_handle, lcpuid, *pmdata, *time)
pm_wp_handle_t wp_handle;
int lcpuid;
pm_data_t *pmdata;
timebasestruct_t *time;

int pm_get_Tdata_lcpu_wp (wp_handle, lcpuid, *pmdata, *times)
pm_wp_handle_t wp_handle;
int lcpuid;
pm_data_t *pmdata;
pm_accu_time_t *times
```

Description

These subroutines return data for only the activities of the processes that belong to a specified workload partition (WPAR).

The specified WPAR handle represents an opaque number that uniquely identifies a WPAR. The **pm_get_wplist** subroutine retrieves this WPAR handle.

The following table shows the information that these subroutines retrieve.

Subroutines	Information
pm_get_data_wp	The current Performance Monitor data for the specified WPAR
pm_get_tdata_wp	• The current Performance Monitor data for the specified WPAR • A timestamp indicating the last time that the hardware counters were read for the specified WPAR
pm_get_Tdata_wp	• The current Performance Monitor data for the specified WPAR • The accumulated time (timebase, PURR time and SPURR time) that the events were counted for the specified WPAR
pm_get_data_lcpu_wp	• The current Performance Monitor data for the specified WPAR and logical processor
pm_get_tdata_lcpu_wp	• The current Performance Monitor data for the specified WPAR and logical processor • A timestamp indicating the last time that the hardware counters were read
pm_get_Tdata_lcpu_wp	• The current Performance Monitor data for the specified WPAR and logical processor • The accumulated time (timebase, PURR time and SPURR time) that the events were counted

The **pm_get_data_lcpu_wp**, **pm_get_tdata_lcpu_wp**, and **pm_get_Tdata_lcpu_wp** subroutines retrieve the current Performance Monitor data for the specified WPAR and logical processor. The specified processor ID represents a value that ranges from 0 through the maximum number that the system defines (with the **_system_configuration.max_ncpus** parameter). The processor ID always represents the same processor, even after Dynamic Reconfiguration operations. If the specified WPAR or logical processor number has never run during the counting interval, the **pm_get_data_lcpu_wp**, **pm_get_tdata_lcpu_wp**, and **pm_get_Tdata_lcpu_wp** subroutines might return an error.

The Performance Monitor data is always a set of 64-bit values, one set per hardware counter on the machines used.

Parameters

Item	Description
lcpuid	The logical processor identifier. Each identifier maintain a link to a particular processor between reboots, even after the Dynamic Reconfiguration. This value must be in the range from 0 through the value of the **_system_configuartion.max_ncpus** parameter.
pmdata	The pointer to a structure that contains the returned Performance Monitor data.
time	The pointer to a structure that contains the *timebase* value the last time that the hardware Performance Monitoring counters were read. This parameter can be converted to time using the **time_base_to_time** subroutine.
times	The pointer to a structure that contains the accumulated time (*timebase*, PURR time, and SPURR time) that the events were counted. Each time counter can be converted to time using the **time_base_to_time** subroutine.
wp_handle	The opaque handle that uniquely identifies a WPAR. This handle can be retrieved from the WPAR name using the **pm_get_wplist** subroutine.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Run the **pm_error** subroutine to decode the error code.

Error Codes

Run the **pm_error** subroutine to decode the error code.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_data_wp_mx, pm_get_tdata_wp_mx, pm_get_data_lcpu_wp_mx, and pm_get_tdata_lcpu_wp_mx Subroutine

Purpose

Returns Performance Monitor data in counter multiplexing mode for a specified workload partition.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_data_wp_mx (wp_handle, *pmdata) pm_wp_handle_t wp_handle;
pm_data_mx_t *pmdata;
```

```
int pm_get_tdata_wp_mx (wp_handle, pmdata, *time) pm_wp_handle_t wp_handle;
pm_data_mx_t *pmdata;
timebasestruct_t *time;

int pm_get_data_lcpu_wp_mx (wp_handle, lcpuid, *pmdata) pm_wp_handle_t
wp_handle;
int lcpuid;
pm_data_mx_t *pmdata;

int pm_get_tdata_lcpu_wp_mx (wp_handle, lcpuid, *pmdata, *time) pm_wp_handle_t
wp_handle;
int lcpuid;
pm_data_mx_t *pmdata;
timebasestruct_t *time;
```

Description

These subroutines return data for only the activities of the processes that belong to a specified workload partition (WPAR).

The specified WPAR handle represents an opaque number that uniquely identifies a WPAR. This WPAR handle can be retrieved using the **pm_get_wplist** subroutine (“pm_get_wplist Subroutine” on page 1299).

The following table shows the information that these subroutines retrieve.

Subroutines	Information
pm_get_data_wp_mx	The current Performance Monitor data in counter multiplexing mode for the specified WPAR
pm_get_tdata_wp_mx	• The current Performance Monitor data in counter multiplexing mode • A timestamp indicating the last time that the hardware counters were read for the specified WPAR
pm_get_data_lcpu_wp_mx	• The current Performance Monitor data in counter multiplexing mode for the specified WPAR and logical processor
pm_get_tdata_lcpu_wp_mx	• The current Performance Monitor data in counter multiplexing mode for the specified WPAR and logical processor • A timestamp indicating the last time that the hardware counters were read for the specified WPAR

The **pm_get_data_lcpu_wp_mx** and the **pm_get_tdata_lcpu_wp_mx** subroutines retrieve the current Performance Monitor data for a specified WPAR and logical processor. The specified processor ID represents a value that ranges from 0 to the value of the **_system_configuration.max_ncpus** parameter. This value always represents the same processor, even after Dynamic Reconfiguration operations. These subroutines might return an error if the specified WPAR or logical processor number has never run during the counting interval.

The Performance Monitor data is always an array of a set of 64-bit values, one per hardware counter on the machines that are used.

The user application must free the array that is allocated to store the accumulated counts and times (the **accu_set** field of the *pmdata* parameter).

Parameters

Item	Description
lcpuid	The logical processor identifier. Each identifier maintains a link to a particular processor between reboots, even after Dynamic Reconfiguration operations. This value must be in the range from 0 through the value of the **_system_configuartion.max_ncpus** parameter.
pmdata	The pointer to a structure that contains the returned Performance Monitor data. The data can be the array of accumulated counters, accumulated time and accumulated PURR and SPURR time for each event set counted.
time	The pointer to a structure containing the **timebase** value that the last time the hardware Performance Monitoring counters were read. This can be converted to time using the **time_base_to_time** subroutine.
wp_handle	The opaque handle that uniquely identifies a WPAR. This handle can be retrieved from the WPAR name using the **pm_get_wplist** subroutine.

Return Values

Item	Description
0	The operation is completed successfully.
Positive error code	Run the **pm_error** subroutine (“pm_error Subroutine” on page 1246) to decode the error.

Errors

Run the **pm_error** subroutine to decode the error code.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_proctype Subroutine

Purpose

Returns the current process type.

Library

Performance Monitor APIs (`libpmapi.a`)

Syntax

```
#include <pmapi.h>

int pm_get_proctype ()
```

Description

The `pm_get_proctype` subroutine returns the current processor type. This value is the same as the one returned in the *proctype* parameter by the `pm_initialize` subroutine.

Return Values

Item	Description
Positive value	Current processor type.
-1	Unsupported processor type.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program Subroutine

Purpose

Retrieves systemwide Performance Monitor settings.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program ( *prog)
pm_prog_t *prog;
```

Description

The **pm_get_program** subroutine retrieves the current systemwide Performance Monitor settings. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the **pm_init** subroutine.

The counting mode includes user mode, the kernel mode, the current counting state, and the process tree mode. If the process tree mode is on, the counting applies only to the calling process and its decendants.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

If the events are represented by a group ID, then the **is_group** bit is set in the mode, and the first element of the events array contains the group ID. The other elements of the events array are not meaningful.

Parameters

Item	Description
prog	Returns which Performance Monitor events and modes are set. Supported modes are: **PM_USER** Counting processes running in user mode **PM_KERNEL** Counting processes running in kernel mode **PM_COUNT** Counting is on **PM_PROCTREE** Counting applies only to the calling process and its descendants

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_group Subroutine

Purpose

Retrieves the Performance Monitor settings for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_group ( pid,  tid,  *prog)
pid_t pid;
tid_t tid;
pm_prog_t *prog;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_get_program_pgroup** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_get_program_pgroup** subroutine with a *ptid* parameter equal to 0.

The **pm_get_program_group** subroutine retrieves the Performance Monitor settings for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the **pm_init** subroutine.

The counting mode includes the user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

Parameters

Item	Description
pid	Process identifier of target thread. The target process must be an argument of a debug process.
tid	Thread identifier of the target thread.
**prog*	Returns which Performance Monitor events and modes are set. Supported modes are: **PM_USER** Counting process running in user mode **PM_KERNEL** Counting process running kernel mode **PM_COUNT** Counting is on **PM_PROCESS** Process level counting group

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_group_mx and pm_get_program_group_mm Subroutines

Purpose

Retrieves the Performance Monitor settings in counter multiplexing mode and multi-mode for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_group_mx ( pid,  tid,  *prog)
pid_t pid;
tid_t tid;
pm_prog_mx_t *prog;

int pm_get_program_group_mm ( pid, tid, *prog_mm)
pid_t pid;
tid_t tid;
pm_prog_mm_t *prog_mm;
```

Description

These subroutines support only the 1:1 threading model. They have been superseded respectively by the **pm_get_program_pgroup_mx** subroutine and the `pm_get_program_pgroup_mm` subroutine, which support both the 1:1 and the M:N threading models. A call to the `pm_get_program_group_mx` subroutine or the `pm_get_program_group_mm` subroutine is respectively equivalent to a call to the **pm_get_program_pgroup_mx** subroutine or the `pm_get_program_pgroup_mm` subroutine with a *ptid* parameter equal to 0.

The **pm_get_program_group_mx** subroutine and the `pm_get_program_group_mm` subroutine retrieve the Performance Monitor settings for the counting group to which a target kernel thread belongs. The thread must be stopped and must be part of a debuggee process under the control of the calling process. This includes mode information and the events being counted, which are in an array of lists of event identifiers. The identifiers come from the lists returned by the **pm_initialize** subroutine.

When counting in multiplexing mode (`pm_get_program_group_mx`), the mode is global to all of the events lists. When counting in multi-mode (`pm_get_program_group_mm`), a mode is associated with each event list.

Counting mode includes the user mode, the kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

The user application must free the allocated array to store the event lists (the events_set field in the *prog* parameter).

Parameters

Item	Description
pid	Process identifier of the target thread. The target process must be an argument of a debug process.
tid	Thread identifier of the target thread.

Item	Description
**prog*	Returns which Performance Monitor events and modes are set. It supports the following modes: **PM_USER** Counting process running in User Mode. **PM_KERNEL** Counting process running in Kernel Mode. **PM_COUNT** Counting is On. **PM_PROCESS** Process level counting group.
**prog_mm*	Returns which Performance Monitor events and associated modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is on. **PM_PROCTREE** Counting that applies only to the calling process and its descendants. The *PM_PROCTREE* mode and the *PM_COUNT* mode are common to all modes set.

Return Values

Item	Description
0	No errors occurred.
Positive error code	See the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

See the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_mx and pm_get_program_mm Subroutines

Purpose

Retrieves system wide Performance Monitor settings in counter multiplexing mode and in multi-mode.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_mx ( *prog)
pm_prog_mx_t *prog;

int pm_get_program_mm (*prog_mm)
pm_prog_mm_t *prog_mm;
```

Description

The **pm_get_program_mx** and pm_get_program_mm subroutines retrieve the current system wide Performance Monitor settings. This includes mode information and the events being counted, which are in an array of list of event identifiers. The identifiers come from the lists returned by the **pm_initialize** subroutine. When you use the pm_get_program_mm subroutine for multi-mode counting, a mode is associated to each event list.

The counting mode includes the user mode, the kernel mode, the current counting state, and the process tree mode. If the process tree mode is set, the counting applies only to the calling process and its descendants.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

If the events are represented by a group ID, then the **is_group** bit is set in the mode, and the first element of each events array contains the group ID. The other elements of the events array are not used.

The user application must free the array allocated to store the event lists (events_set field in prog).

Parameters

Item	Description
prog	Returns which Performance Monitor events and modes are set. It supports the following modes: **PM_USER** Counting processes running in the user mode. **PM_KERNEL** Counting processes running in the kernel mode. **PM_COUNT** Counting is on. **PM_PROCTREE** Counting applies only to the calling process and its descendants.

Item	Description
prog_mm	Returns which Performance Monitor events and associated modes are set. It supports the following modes: **PM_USER** Counting processes running in the user mode. **PM_KERNEL** Counting processes running in the kernel mode. **PM_COUNT** Counting is On. **PM_PROCTREE** Counting applies only to the calling process and its descendants. The *PM_PROCTREE* mode and the *PM_COUNT* mode are common to all mode set.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the pm_error (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_mygroup Subroutine

Purpose

Retrieves the Performance Monitor settings for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_mygroup ( *prog)
pm_prog_t *prog;
```

Description

The **pm_get_program_mygroup** subroutine retrieves the Performance Monitor settings for the counting group to which the calling kernel thread belongs. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the **pm_init** subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

Parameters

Item	Description
**prog*	Returns which Performance Monitor events and modes are set. Supported modes are: **PM_USER** Counting processes running in user mode **PM_KERNEL** Counting processes running in kernel mode **PM_COUNT** Counting is on **PM_PROCESS** Process level counting group

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_mygroup_mx and pm_get_program_mygroup_mm Subroutines

Purpose

Retrieves the Performance Monitor settings in counter multiplexing mode and multi-mode for the counting group to which the calling thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_mygroup_mx ( *prog)
pm_prog_mx_t *prog;

int pm_get_program_mygroup_mm (*prog_mm)
pm_prog_mm_t *prog_mm;
```

Description

The **pm_get_program_mygroup_mx** and the pm_get_program_mygroup_mm subroutines retrieve the Performance Monitor settings for the counting group to which the calling kernel thread belongs. This includes mode information and the events being counted, which are in an array of lists of event identifiers. The identifiers come from the lists returned by the **pm_initialize** subroutine.

When counting in multiplexing mode, the mode is global to all of the events lists. When counting in multi-mode, a mode is associated to each event list.

Counting mode includes the user mode, the kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

The user application must free the allocated array to store the event lists (the events_set field in the *prog* parameter).

Parameters

Item	Description
**prog*	Returns which Performance Monitor events and modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is on. **PM_PROCESS** Process level counting group.
**prog_mm*	Returns which Performance Monitor events and associated modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is On. **PM_PROCTREE** Counting applies only to the calling processes and its descendants. The *PM_PROCTREE* mode and the *PM_COUNT* mode are common to all modes set.

Return Values

Item	Description
0	No errors occurred.
Positive error code	See the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

See the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_mythread Subroutine

Purpose

Retrieves the Performance Monitor settings for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_mythread ( *prog)
pm_prog_t *prog;
```

Description

The **pm_get_program_mythread** subroutine retrieves the Performance Monitor settings for the calling kernel thread. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the **pm_init** subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

Parameters

Item	Description
**prog*	Returns which Performance Monitor events and modes are set. Supported modes are: **PM_USER** Counting processes running in user mode **PM_KERNEL** Counting processes running in kernel mode **PM_COUNT** Counting is on

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_mythread_mx and pm_get_program_mythread_mm Subroutines

Purpose

Retrieves the Performance Monitor settings in counter multiplexing mode and multi-mode for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_mythread_mx (*prog)
pm_prog_mx_t *prog;

int pm_get_program_mythread_mm (*prog_mm)
pm_prog_mm_t *prog_mm;
```

Description

The **pm_get_program_mythread_mx** and the `pm_get_program_mythread_mm` subroutines retrieve the Performance Monitor settings for the calling kernel thread. This includes mode information and the events being counted, which are in an array of lists of event identifiers. The event identifiers come from the lists returned by the **pm_initialize** subroutine.

When counting in multiplexing mode, the mode is global to all of the events lists. When counting in multi-mode, a mode is associated with each event list.

Counting mode includes the user mode, the kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

The user application must free the allocated array to store the event lists (the events_set field in the *prog* parameter).

Parameters

Item	Description
**prog*	Returns which Performance Monitor events and modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is On.
**prog_mm*	Returns which Performance Monitor events and associated modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is On. **PM_PROCTREE** Counting that applies only to the calling processes and its descendants. The *PM_PROCTREE* mode and the *PM_COUNT* mode are common to all modes set.

Return Values

Item	Description
0	No errors occurred.
Positive error code	See the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

See the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_pgroup Subroutine

Purpose

Retrieves Performance Monitor settings for the counting group to which a target pthread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_pgroup ( pid,  tid,  ptid,  *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_t *prog;
```

Description

The **pm_get_program_pgroup** subroutine retrieves the Performance Monitor settings for the counting group to which a target pthread belongs. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the **pm_inititialize** subroutine.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the*ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The counting mode includes the user mode and kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the **pm_initialize** subroutine), a threshold value is also returned.

Parameters

Item	Description
pid	Process ID of target pthread. The target process must be an argument of a debug process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
*prog	Returns which Performance Monitor events and modes are set. The following modes are supported: **PM_USER** Counts process running in user mode **PM_KERNEL** Counts process running kernel mode **PM_COUNT** Counting is on **PM_PROCESS** Process-level counting group

Return Values

Item	Description
`0`	No errors occurred.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_pgroup_mx and pm_get_program_pgroup_mm Subroutines

Purpose

Retrieves Performance Monitor settings in counter multiplexing mode and multi-mode for the counting group to which a target pthread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_pgroup_mx ( pid,  tid,  ptid,  *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_mx_t *prog;

int pm_get_program_pgroup_mm ( pid, tid, ptid, prog_mm)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_mm_t *prog_mm;
```

Description

The **pm_get_program_pgroup_mx** and the `pm_get_program_pgroup_mm` subroutine retrieve the Performance Monitor settings for the counting group to which a target pthread belongs. The pthread must be stopped and must be part of a debuggee process, which is under the control of the calling process. This includes mode information and the events being counted, which are in an array of lists of event identifiers. The event identifiers come from the lists returned by the **pm_inititialize** subroutine.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with the *tid* parameter specified.

When counting in multiplexing mode, the mode is global to all of the events lists. When counting in the multi-mode, a mode is associated with each event list.

The counting mode includes the user mode and kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the **pm_initialize** subroutine), a threshold value is also returned.

The user application must free the allocated array to store the event lists (the events_set field in the *prog* parameter).

Parameters

Item	Description
pid	Process ID of target pthread. The target process must be an argument of a debug process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**prog*	Returns which Performance Monitor events and modes are set. It supports the following modes: **PM_USER** Counts process running in User Mode. **PM_KERNEL** Counts process running Kernel Mode. **PM_COUNT** Counting is On. **PM_PROCESS** Process-level counting group.
**prog_mm*	Returns which Performance Monitor events and associated modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is On. **PM_PROCTREE** Counting applies only to the calling processes and its descendants. The *PM_PROCTREE* mode and the *PM_COUNT* mode are common to all modes set.

Return Values

Item	Description
`0`	No errors occurred.
`Positive error code`	See the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

See the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_pthread Subroutine

Purpose

Retrieves the Performance Monitor settings for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_pthread ( pid,  tid,  ptid,  *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_t *prog;
```

Description

The **pm_get_program_pthread** subroutine retrieves the Performance Monitor settings for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers must be selected from the lists returned by the **pm_inititialize** subroutine.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the **pm_initialize** subroutine), a threshold value is also returned.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be an argument of a debug process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
*prog	Returns which Performance Monitor events and modes are set. The following modes are supported: **PM_USER** Counts processes running in User Mode **PM_KERNEL** Counts processes running in Kernel Mode **PM_COUNT** Counting is On

Return Values

Item	Description
`0`	No errors occurred.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_pthread_mx and pm_get_program_pthread_mm Subroutines

Purpose

Retrieves the Performance Monitor settings in counter multiplexing mode and multi-mode for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_pthread_mx ( pid,  tid,  ptid,  *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_mx_t *prog;

int pm_get_program_pthread_mm ( pid, tid, ptid, prog_mm)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_mm_t *prog_mm;
```

Description

The **pm_get_program_pthread_mx** and the `pm_set_program_pthread_mm` subroutines retrieve the Performance Monitor settings for a target pthread. The pthread must be stopped and must be part of a debuggee process, that is under the control of the calling process. This includes mode information and the events being counted, which are in an array of lists of event identifiers. The event identifiers must be selected from the lists returned by the **pm_inititialize** subroutine.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

When counting in multiplexing mode, the mode is global to all of the events lists. When counting in the multi-mode, a mode is associated with each event list.

Counting mode includes the user mode, the kernel mode, and the current counting state.

If the list includes an event that can be used with a threshold (as indicated by the **pm_initialize** subroutine), a threshold value is also returned.

The user application must free the allocated array to store the event lists (the events_set field in the *prog* parameter).

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be an argument of a debug process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**prog*	Returns which Performance Monitor events and modes are set. It supports the following modes: **PM_USER** Counts processes running in User Mode. **PM_KERNEL** Counts processes running in Kernel Mode. **PM_COUNT** Counting is On.
**prog_mm*	Returns which Performance Monitor events and associated modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is On. **PM_PROCTREE** Counting that applies only to the calling processes and its descendants. The *PM_PROCTREE* mode and the *PM_COUNT* mode are common to all modes set.

Return Values

Item	Description
`0`	No errors occurred.
`Positive error code`	See the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

See the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_thread Subroutine

Purpose

Retrieves the Performance Monitor settings for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_thread ( pid,  tid,  *prog)
pid_t pid;
tid_t tid;
pm_prog_t *prog;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_get_program_pthread** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_get_program_pthread** subroutine with a *ptid* parameter equal to 0.

The **pm_get_program_thread** subroutine retrieves the Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process. This includes mode information and the events being counted, which are in a list of event identifiers. The identifiers come from the lists returned by the **pm_init** subroutine.

The counting mode includes user mode and kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

Parameters

Item	Description
pid	Process identifier of the target thread. The target process must be an argument of a debug process.
tid	Thread identifier of the target thread.

Item	Description
**prog*	Returns which Performance Monitor events and modes are set. Supported modes are: **PM_USER** Counting processes running in **User** mode **PM_KERNEL** Counting processes running in **Kernel** mode **PM_COUNT** Counting is On

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_thread_mx and pm_get_program_thread_mm Subroutines

Purpose

Retrieves the Performance Monitor settings in counter multiplexing mode and multi-mode for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_thread_mx ( pid,  tid,  *prog)
pid_t pid;
tid_t tid;
pm_prog_mx_t *prog;

int pm_get_program_thread_mm (pid, tid, *prog_mm)
pid_t pid;
tid_t tid;
pm_prog_mm_t *prog_mm;
```

Description

These subroutines support only the 1:1 threading model. They have been superseded respectively by the **pm_get_program_pthread_mx** and the `pm_get_program_pthread_mm` subroutines, which support both the 1:1 and the M:N threading models. A call to the `pm_get_program_thread_mx` subroutine or to the `pm_get_program_thread_mm` subroutine is respectively equivalent to a call to the **pm_get_program_pthread_mx** subroutine or the `pm_get_program_pthread_mm` subroutine with a *ptid* parameter equal to 0.

The **pm_get_program_thread_mx** subroutine and the `pm_get_program_thread_mm` subroutine retrieve the Performance Monitor settings for a target kernel thread. The thread must be stopped and must be part of a debuggee process under the control of the calling process. This includes mode information and the events being counted, which are in an array of list of event identifiers. The event identifiers come from the lists returned by the **pm_initialize** subroutine.

When counting in multiplexing mode, the mode is global to all of the events lists. When counting in multi-mode, a mode is associated to each event list.

Counting mode includes the user mode, the kernel mode, and the current counting state.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value is also returned.

The user application must free the allocated array to store the event lists (the events_set field in the *prog* parameter).

Parameters

Item	Description
pid	Process identifier of the target thread. The target process must be an argument of a debug process.
tid	Thread identifier of the target thread.
**prog*	Returns which Performance Monitor events and modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is On.
**prog_mm*	Returns which Performance Monitor events and associated modes are set. It supports the following modes: **PM_USER** Counting processes running in User Mode. **PM_KERNEL** Counting processes running in Kernel Mode. **PM_COUNT** Counting is On. **PM_PROCTREE** Counting that applies only to the calling process and its descendants. The *PM_PROCTREE* mode and the *PM_COUNT* mode are common to all modes set.

Return Values

Item	Description
0	No errors occurred.
Positive error code	See the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

See the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_wp Subroutine

Purpose

Retrieves system-wide Performance Monitor setting for a specified workload partition (WPAR).

Library

Performance Monitor APIs Library (**libpmapi.a**).

Syntax

```
#include <pmapi.h>
int pm_get_program_wp (cid, *prog)
cid_t cid;
pm_prog_t *prog;
```

Description

The **pm_get_program_wp** subroutine retrieves system-wide Performance Monitor settings for the processes that belong to the specified workload partition. These settings include the mode information and the events that are being counted.

The events being counted are in a list of event identifiers. The identifiers must be selected from the list that the **pm_init** subroutine returns. If the list includes an event that can be used with a threshold, you can specify a threshold value.

If the events are represented by a group ID, then the **is_group** bit is set in the mode, and the first element of the events array contains the group ID. The other elements of the events array are not meaningful.

The counting mode includes both User mode and Kernel mode, or either of them; the Initial Counting state; and the Process Tree mode.

If the Process Tree mode is set to the On state, the counting only applies to the calling process and its descendants.

Parameters

Item	Description
cid	Specifies the identifier of the WPAR for which the subroutine is to retrieve. The CID can be obtained from the WPAR name using the **getcorralid** system call.
prog	Returns the Performance Monitor events and modes that are set. The following modes are supported: **PM_USER** Counting the processes that are running in User mode. **PM_KERNEL** Counting the processes that are running in Kernel mode. **PM_COUNT** The counting is on. **PM_PROCTREE** Counting only the calling process and its descendants.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Run the **pm_error** subroutine to decode the error code.

Error Codes

To decode the error code, see the **pm_error** subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_program_wp_mm Subroutine

Purpose

Returns Performance Monitor settings in counter multiplexing mode for a specified Workload partition.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_get_program_wp_mm (cid, *prog_mm)
cid_t cid;
pm_prog_mm_t *prog_mm;
```

Description

The **pm_get_program_wp_mm** subroutine retrieves the current Performance Monitor settings in counter multiplexing mode for a specified workload partition (WPAR). The settings include the mode information and the events being counted, which are in an array of a list of event identifiers. The identifiers must be selected from the lists that the "pm_initialize Subroutine" on page 1302 subroutine returns. If the list includes an event that can be used with a threshold, a threshold value is also returned.

When you use the **pm_get_program_wp_mm** subroutine for multi-mode counting, a mode is associated to each event list.

The counting mode includes both User mode and Kernel mode, or either of them; the current Counting state; and the Process Tree mode. If the Process Tree mode is set, the counting is applied to only the calling process and its descendants.

If the events are represented by a group ID, then the **is_group** bit is set in the mode, and the first element of each events array contains the group ID. The other elements of the events array are not used.

The user application must free the array allocated to store the event lists.

Parameters

Item	Description
cid	Specifies the identifier of the WPAR for which the programming is to be retrieved. The CID can be obtained from the WPAR name using the **getcorralid** system call.
prog_mm	Returns the Performance Monitor events and modes that are set. The following modes are supported: **PM_USER** Counting the processes that are running in User mode. **PM_KERNEL** Counting the processes that are running in Kernel mode. **PM_COUNT** The counting is on. **PM_PROCTREE** Counting only the activities of the calling process and its descendants. The **PM_PROCTREE** mode and the **PM_COUNT** mode are common to all mode set.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Run the **pm_error** subroutine ("pm_error Subroutine" on page 1246) to decode the error code.

Error Codes

To decode the error code, see the **pm_error** subroutine ("pm_error Subroutine" on page 1246).

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_get_wplist Subroutine

Purpose

Retrieves the list of available workload partition contexts for Performance Monitoring.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>int pm_get_wplist (*name, *wp_list, *size)
const char *name;
pm_wpar_ctx_info_t *wp_list;
int *size;
```

Description

The **pm_get_wplist** subroutine retrieves information on the workload partitions (WPAR) that are active during the last system-wide counting. This information includes the CID, name, and opaque handle of the WPAR. With the **pm_get_data_wp** or **pm_get_data_wp_mx** subroutines, the handle can retrieve system-wide Performance Monitor data for a specified WPAR.

If the *name* parameter is specified, the **pm_get_wplist** subroutine retrieves information for only the specified WPAR. Otherwise, the **pm_get_wplist** subroutine retrieves information for all WPAR that are active during the last system-wide counting.

If the *wp_list* parameter is not specified, the **pm_get_wplist** subroutine only returns the number of available WPAR contexts in that the *size* parameter points to. Otherwise, the array that the *wp_list* parameter points to is filled with up to the number of WPAR contexts that the *size* parameter defines.

The **pm_get_wplist** subroutine can allocate a *wp_list* array large enough to store all available WPAR contexts. To do this, calls the **pm_get_wplist** subroutine twice. The first call will retrieve the number of available WPAR contexts only.

Note: It is suggested to call the **pm_get_wplist** subroutine while no counting is active, because WPAR contexts can be created dynamically during an active counting.

On output to the **pm_get_wplist** subroutine, the variable that the *size* parameter points to is set to the number of available WPAR contexts for Performance Monitoring.

Parameters

Item	Description
name	The name of the WPAR for which information is to be retrieved. If the *name* is not specified, information for all WPAR that are active during the last system-wide counting is retrieved.
size	Pointer to a variable that contains the number of elements of the array that the **wp_list** parameter points to. On output, this variable will be filled with the actual number of WPAR contexts available.
wp_list	Pointer to an array that will be filled with WPAR contexts. If the *wp_list* parameter is not specified, only the number of WPAR contexts is to be retrieved.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Run the **pm_error** subroutine (“pm_error Subroutine” on page 1246) to decode the error code.

Error Codes

Run the **pm_error** subroutine to decode the error code.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_init Subroutine

Purpose

Initializes the Performance Monitor APIs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_init ( filter, *pminfo, *pm_groups_info)
int filter;
pm_info_t *pminfo;
pm_groups_info_t *pm_groups_info;
```

Description

Note: The **pm_init** subroutine cannot be used on processors newer than POWER4. With such processors, the **pm_initialize** subroutine must be used.

The **pm_init** subroutine initializes the Performance Monitor API library. It returns, after taking into account a *filter* on the status of the events, the number of counters available on this processor, and one table per counter with the list of events available. For each event, an event identifier, a status, a flag indicating if the event can be used with a threshold, two names, and a description are provided.

The event identifier is used with all the **pm_set_program** interfaces and is also returned by all of the **pm_get_program** interfaces. Only event identifiers present in the table returned can be used. In other words, the *filter* is effective for all API calls.

The status describes whether the event has been verified, is still unverified, or works with some caveat, as explained in the description. This field is necessary because the filter can be any combination of the three available status bits. The flag points to events that can be used with a threshold.

Only events categorized as *verified* have gone through full verification. Events categorized as *caveat* have been verified only within the limitations documented in the event description. Events categorized as *unverified* have undefined accuracy. Use caution with *unverified* events; the Performance Monitor software is essentially providing a service to read hardware registers which may or may not have any meaningful

content. Users may experiment with unverified event counters and determine for themselves what, if any, use they may have for specific tuning situations.

The short mnemonic name is provided for easy keyword-based search in the event table (see the sample program **/usr/samples/pmapi/sysapit2.c** for code using mnemonic names). The complete name of the event is also available and a full description for each event is returned.

The structure returned also has the threshold multiplier for this processor and the processor name

On some platforms, it is possible to specify event groups instead of individual events. Event groups are predefined sets of events. Rather than specify each event individually, a single group ID is specified. On some platforms, such as POWER4, use of the event groups is required, and attempts to specify individual events return an error.

The interface to **pm_init** has been enhanced to return the list of supported event groups in an optional third parameter. For binary compatibilty, the third parameter must be explicitly requested by OR-ing the bitflag, PM_GET_GROUPS, into the *filter* parameter.

If the *pm_groups_info* parameter returned by **pm_init** is NULL, there are no supported event groups for the platform. Otherwise an array of **pm_groups_t** structures are returned in the **event_groups** field. The length of the array is given by the **max_groups** field.

The **pm_groups_t** structure contains a group identifier, two names and a description that are similar to those of the individual events. In addition, there is an array of integers that specify the events contained in the group.

Parameters

Item	Description
filter	Specifies which event types to return. **PM_VERIFIED** Events which have been verified **PM_UNVERIFIED** Events which have not been verified **PM_CAVEAT** Events which are usable but with caveats as described in the long description
**pminfo*	Returned structure with processor name, threshold multiplier, and a filtered list of events with their current status.
**pm_groups_info*	Returned structure with list of supported groups. This parameter is only meaningful if PM_GET_GROUPS is OR-ed into the *filter* parameter.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

See the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_initialize Subroutine

Purpose

Initializes the Performance Monitor APIs and returns information about a processor.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_initialize ( filter,  *pminfo, *pmgroups, proctype)
int filter;
pm_info2_t *pminfo;
pm_groups_info_t *pmgroups;
int proctype;
```

Description

The **pm_initialize** subroutine initializes the Performance Monitor API library and retrieves information about a type of processor (if the specified *proctype* is not **PM_CURRENT**). It takes into account a *filter* on the events status, then it returns the number of counters available on this processor and one table per counter containing the list of available events. For each event, it provides an event identifier, a status, two names, and a description. The status contains a set of flags indicating: the event status, if the event can be used with a threshold, if the event is a shared event, and if the event is a grouped-only event.

The event identifier is used with all **pm_set_program** interfaces and is also returned by all of the **pm_get_program** interfaces. Only event identifiers present in the returned table can be used. In other words, the *filter* is effective for all API calls.

The status describes whether the event has been verified, is still unverified, or works with some caveat, as explained in the description. This field is necessary because the filter can be any combination of the three available status bits. The flag points to events that can be used with a threshold.

Only events categorized as verified have been fully verified. Events categorized as *caveat* have been verified only with the limitations documented in the event description. Events categorized as *unverified* have an undefined accuracy. Use *unverified* events cautiously; the Performance Monitor software provides essentially a service to read hardware registers, which might or might not have meaningful content. Users might experiment for themselves with unverified event counters to determine if they can be used for specific tuning situations.

The short mnemonic name is provided for an easy keyword-based search in the event table (see the sample program **/usr/samples/pmapi/cpi.c** for code using mnemonic names). The complete name of the event is also available, and a full description for each event is returned.

The returned structure also contains the threshold multipliers for this processor, the processor name, and its characteristics. On some platforms, up to three threshold multipliers are available.

On some platforms, it is possible to specify event groups instead of individual events. Event groups are predefined sets of events. Rather than specify each event individually, a single group ID is specified. On some platforms, such as POWER4, using event groups is mandatory, and specifying individual events returns an error.

The interface to **pm_initialize** returns the list of supported event groups in its third parameter. If the *pmgroups* parameter returned by **pm_initialize** is NULL, there are no supported event groups for the platform. Otherwise an array of **pm_groups_t** structures is returned in the **event_groups** field. The length of the array is given by the **max_groups** field.

The **pm_groups_t** structure contains a group identifier, two names, and a description that are all similar to those of the individual events. In addition, an array of integers specifies the events contained in the group.

If the *proctype* parameter is not set to **PM_CURRENT**, the Performance Monitor APIs library is not initialized, and the subroutine only returns information about the specified processor and those events and groups available in its parameters (*pminfo* and *pmgroups*) taking into account the filter. If the *proctype* parameter is set to **PM_CURRENT**, in addition to returning the information described, the Performance Monitor APIs library is initialized and ready to accept other calls.

Parameters

Item	Description
filter	Specifies which event types to return. **PM_VERIFIED** Events that have been verified. **PM_UNVERIFIED** Events that have not been verified. **PM_CAVEAT** Events that are usable but with caveats, as explained in the long description.
pmgroups	Returned structure containing the list of supported groups.
pminfo	Returned structure containing the processor name, the threshold multiplier and a filtered list of events with their current status.
proctype	Initializes the Performance Monitor API and retrieves information about a specific processor type: **PM_CURRENT** Retrieves information about the current processor and initializes the Performance Monitor API library. **other** Retrieves information about a specific processor.

Return Values

Item	Description
0	No errors occurred.
Positive error code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_reset_data and pm_reset_data_wp Subroutines

Purpose

Resets system-wide Performance Monitor data.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_reset_data ()int pm_reset_data_wp (cid_t cid)
```

Description

The **pm_reset_data** subroutine resets the current system-wide Performance Monitor data. The **pm_reset_data_wp** subroutine resets the system-wide Performance Monitor data for a specified workload partition (WPAR).

The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0.

Parameters

Item	Description
cid	Specifies the identifier of the WPAR that the subroutine deletes. The CID can be obtained from the WPAR name using the **getcorralid** subroutine.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

See the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_reset_data_group Subroutine

Purpose

Resets Performance Monitor data for a target thread and the counting group to which it belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_reset_data_group ( pid,  tid)
pid_t pid;
tid_t tid;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_reset_data_pgroup** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_reset_data_pgroup** subroutine with a *ptid* parameter equal to 0.

The **pm_reset_data_group** subroutine resets the current Performance Monitor data for a target kernel thread and the counting group to which it belongs. The thread must be stopped and must be part of a debugee process, under control of the calling process. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0. Because the data for all the other threads in the group is not affected, the group is marked as inconsistent unless it has only one member.

Parameters

Item	Description
pid	Process ID of target thread. Target process must be a debuggee of the caller process.
tid	Thread ID of target thread.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_reset_data_mygroup Subroutine

Purpose

Resets Performance Monitor data for the calling thread and the counting group to which it belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_reset_data_mygroup()
```

Description

The **pm_reset_data_mygroup** subroutine resets the current Performance Monitor data for the calling kernel thread and the counting group to which it belongs. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0. Because the data for all the other threads in the group is not affected, the group is marked as inconsistent unless it has only one member.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_reset_data_mythread Subroutine

Purpose

Resets Performance Monitor data for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_reset_data_mythread()
```

Description

The **pm_reset_data_mythread** subroutine resets the current Performance Monitor data for the calling kernel thread. The data is a set (one per hardware counter on the machine) of 64-bit values. All values are reset to 0.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_reset_data_pgroup Subroutine

Purpose

Resets Performance Monitor data for a target pthread and the counting group to which it belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_reset_data_pgroup ( pid,  tid,  ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;
```

Description

The **pm_reset_data_pgroup** subroutine resets the current Performance Monitor data for a target pthread and the counting group to which it belongs. The pthread must be stopped and must be part of a debugee process, under control of the calling process. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0. Because the data for all the other pthreads in the group is not affected, the group is marked as inconsistent unless it has only one member.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.

Item	Description
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_reset_data_pthread Subroutine

Purpose

Resets Performance Monitor data for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_reset_data_pthread ( pid,  tid,  ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;
```

Description

The **pm_reset_data_pthread** subroutine resets the current Performance Monitor data for a target pthread. The pthread must be stopped and must be part of a debuggee process. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_reset_data_thread Subroutine

Purpose

Resets Performance Monitor data for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_reset_data_thread ( pid,  tid)
pid_t pid;
tid_t tid;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_reset_data_pthread** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_reset_data_pthread** subroutine with a *ptid* parameter equal to 0.

The **pm_reset_data_thread** subroutine resets the current Performance Monitor data for a target kernel thread. The thread must be stopped and must be part of a debuggee process. The data is a set (one per hardware counter on the machine used) of 64-bit values. All values are reset to 0.

Parameters

Item	Description
pid	Process id of target thread. Target process must be a debuggee of the caller process.
tid	Thread id of target thread.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, datatypes, and subroutines.

pm_set_counter_frequency_pthread, pm_set_counter_frequency_thread, or pm_set_counter_frequency_mythread Subroutine

Purpose

Configures the counter frequencies for the target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int pm_set_counter_frequency_pthread (pid_t pid, tid_t tid,
 ptid_t ptid,
 unsigned counter_freq [MAX_COUNTERS])

int pm_set_counter_frequency_thread (pid_t pid, tid_t tid,
 unsigned counter_freq [MAX_COUNTERS])

int pm_set_counter_frequency_mythread (unsigned counter_freq [MAX_COUNTERS])
```

Description

The **pm_set_counter_frequency_pthread**, **pm_set_counter_frequency_thread**, or **pm_set_counter_frequency_mythread** subroutines configure the counter frequency values in the Performance Monitor Counters (PMCs) for a given thread.

The **pm_set_counter_frequency_pthread** subroutine must be used to configure the counter frequency for a target pthread.

The **pm_set_counter_frequency_thread** subroutine must be used to configure the counter frequency for a target kernel thread.

The **pm_set_counter_frequency_mythread** subroutine must be used to configure the counter frequency for self thread.

Parameters

Item	Description
pid	Process ID of the target thread.
tid	Kernel thread ID of the target thread. The value can be set to zero, if the parameter is not required.
ptid	Pthread ID of the target thread. The value can be set to zero if the parameter is not required.
counter_freq	Counter frequencies of the corresponding PMCs.

Return Values

If unsuccessful, a value other than zero is returned and a positive error code is set. If successful, a value of zero is returned.

Error Codes

The subroutine is unsuccessful if the following error codes are returned:

Item	Description
Pmapi_NoInit	The **pm_initialize** subroutine is not called.
Pmapi_NoSetProg	The **pm_set_program** subroutine is not called.
Other non-zero error codes	Returned by the **pmsvcs** subroutine.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

pm_set_ebb_handler Subroutine

Purpose

Configures the Event-Based Branching (EBB) facility for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int pm_set_ebb_handler (void * handler_address, void * data_area,)
```

Description

The **pm_set_ebb_handler** subroutine configures EBB and allows user to specify the effective address (EA) of the next instruction to be run based on the occurrence of specific events. Events and frequencies are configured by the thread before calling this subroutine.

Events can be configured by using the **pm_set_program_mythread**, **pm_set_program_pthread**, or **pm_set_program_thread** subroutine. One of these subroutines must be called before calling the **pm_set_ebb_handler** subroutine. The **pm_set_program_*** subroutines must be called with the *no_inherit* flag.

Counter frequencies can be configured by using the **pm_set_counter_frequency_mythread**, or **pm_set_counter_frequency_pthread**, or **pm_set_counter_frequency_thread** subroutine.

Note:

- The **pm_set_ebb_handler** subroutine can be called only by the thread that is profiling itself (self-profiling threads) and it cannot be called if the thread is part of a group.
- The **pm_set_ebb_handler** subroutine can only be called when the thread mode is 1:1 and when counting for the thread is not started.

Parameters

Item	Description
handler_address	The effective address of the user handler.
data_area	The allocated data area. This data area is accessible from the EBB handler.

Return Values

If unsuccessful, a value other than zero is returned and a positive error code is set. If successful, a value of zero is returned.

Error Codes

The subroutine is unsuccessful if the following error codes are returned:

Item	Description
Pmapi_NoInit	The **pm_initialize** subroutine is not called.
Pmapi_Unsupported_EBBThreadMode	The thread is not running in the 1:1 mode.
Pmapi_NoSetProg	The **pm_set_program** subroutine is not called.
Pmapi_Invalid_EBB_handler_addr	The value of the *handler_address* is NULL.
Pmapi_Invalid_EBB_data_addr	The value of the *data_area* is NULL.
Pmapi_Malloc_Err	The **malloc** subroutine fails while allocating memory to the **pthread_EBB_registration_t** structure.

Item	Description
Pmapi_Invalid_EBB_Config	The *PTHREAD_EBB_PMU_TYPE* flag is not passed to the **pthread** subroutine.
Pmapi_EBB_Already_Exists	The EBB handler is already setup for the thread.
Other non-zero error codes	Returned by the call to the **pmsvcs** subroutine.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

pm_set_program Subroutine

Purpose

Sets system wide Performance Monitor programmation.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program ( *prog)
pm_prog_t *prog;
```

Description

The **pm_set_program** subroutine sets system wide Performance Monitor programmation. The setting includes the events to be counted, and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the **pm_init** subroutine.

The counting mode includes User Mode and/or Kernel Mode, the Initial Counting State, and the Process Tree Mode. The Process Tree Mode sets counting to On only for the calling process and its descendants. The defaults are set to Off for User Mode and Kernel Mode. The initial default state is set to delay counting until the **pm_start** subroutine is called, and to count the activity of all the processes running in the system.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

On some platforms, event groups can be specified instead of individual events. This is done by setting the bitfield **is_group** in the mode, and placing the group ID into the first element of the events array. (The group ID was obtained by **pm_init**).

Parameters

Item	Description
**prog*	Specifies the events and modes to use in Performance Monitor setup. The following modes are supported: ***PM_USER*** Counts processes running in User Mode (default is set to Off) ***PM_KERNEL*** Counts processes running in Kernel Mode (default is set to Off) ***PM_COUNT*** Starts counting immediately (default is set to Not to Start Counting) ***PM_PROCTREE*** Sets counting to On only for the calling process and its descendants (default is set to Off)

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_group Subroutine

Purpose

Sets Performance Monitor programmation for a target thread and creates a counting group.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_group ( pid,  tid,  *prog)
pid_t pid;
```

```
tid_t tid;
pm_prog_t *prog;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_set_program_pgroup** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_set_program_pgroup** subroutine with a *ptid* parameter equal to 0.

The **pm_set_program_group** subroutine sets the Performance Monitor programmation for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the **pm_init** subroutine.

This call also creates a counting group, which includes the target thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the target process.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the **pm_start_group** subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

Parameters

Item	Description
pid	Process ID of target thread. Target process must be a debuggee of a calling process.
tid	Thread ID of target thread.
**prog*	**PM_USER** Counts processes running in User Mode (default is set to Off) **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off) **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting) **PM_PROCESS** Creates a process-level counting group

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_group_mx and pm_set_program_group_mm Subroutines

Purpose

Sets the Performance Monitor program in counter multiplexing mode and multi-mode for a target thread and creates a counting group.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_group_mx ( pid,  tid,  *prog)
pid_t pid;
tid_t tid;
pm_prog_mx_t *prog;

int pm_set_program_group_mm ( pid, tid, *prog_mm)
pid_t pid;
tid_t tid;
pm_prog_mm_t *prog_mm;
```

Description

The `pm_set_program_group_mx` and `pm_set_program_group_mm` subroutines support only the 1:1 threading model. They have been superseded respectively by the **pm_set_program_pgroup_mx** and `pm_set_program_pgroup_mm` subroutines, which support both the 1:1 and the M:N threading models. A call to the `pm_set_program_pgroup_mx` or `pm_set_program_pgroup_mm` subroutine is respectively equivalent to a call to the **pm_set_program_pgroup_mx** or `pm_set_program_pgroup_mm` subroutine with a *ptid* parameter equal to 0.

The **pm_set_program_group_mx** and `pm_set_program_group_mm` subroutines set the Performance Monitor program respectively in counter multiplexing mode or in multi-mode for a target kernel thread. The thread must be stopped and must be part of a debuggee process, which is under the control of the calling process.

The `pm_set_program_group_mx` subroutine setting includes the list of the event arrays to be counted and the mode in which to count. The mode is global to all of the event lists. The events to count are in an array of lists of event identifiers.

The `pm_set_program_group_mm` subroutine setting includes the list of the event arrays to be counted, and the associated mode in which to count each event array. A counting mode is defined for each event array.

The event identifiers must be selected from the lists returned by the **pm_initialize** subroutine.

Both subroutines create a counting group, which includes the target thread and any thread which it, or any of its descendants, will create in the future. The group can also be defined as containing all the existing and future threads belonging to the target process.

The counting mode for the subroutines includes the User Mode, the Kernel Mode, or both of them, and the Initial Counting State. The default is set to Off for the User Mode and the Kernel Mode. The initial default state is set to delay counting until the **pm_start_group** subroutine is called.

When you use the `pm_set_program_group_mm` subroutine for multi-mode counting, the Process Tree Mode and the Start Counting Mode are fixed by their values that are defined in the first programming set.

If the list includes an event that can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

Parameters

Item	Description
pid	Specifies the process ID of target thread. The target process must be a debuggee of a calling process.
tid	Specifies the thread ID of the target thread.
**prog*	Specifies the events and modes to use in the Performance Monitor setup. The *prog* parameter supports the following modes: **PM_USER** Counts processes running in User Mode (default is set to Off). **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to start counting). **PM_PROCESS** Creates a process-level counting group.
* *prog_mm*	Specifies the events and the modes to use in the Performance Monitor setup. The *prog_mm* parameter supports the following modes: **PM_USER** Counts processes running in User Mode (default is set to Off). **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to start counting). **PM_PROCTREE** Sets counting to On only for the calling process and its descendents (default is set to Off). The *PM_PROCTREE* mode and the *PM_COUNT* mode defined in the first setting fix value for the counting.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	See the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

See the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_mx and pm_set_program_mm Subroutines

Purpose

Sets system wide Performance Monitor programmation in counter multiplexing mode and in multi-mode.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_mx (*prog)
pm_prog_mx_t *prog;

int pm_set_program_mm (*prog_mm)
pm_prog_mm_t *prog_mm;
```

Description

The **pm_set_program_mx** and `pm_set_program_mm` subroutines set system wide Performance Monitor programmation in counter multiplexing mode.

The `pm_set_program_mx` setting includes the list of the event arrays to be counted, and a mode in which to count. The events to count are in an array of list of event identifiers. The mode is global to all the event lists.

The `pm_set_program_mm` setting includes the list of the event arrays to be counted, and the associated mode in which to count each event array. A counting mode is defined for each event array.

The identifiers must be selected from the lists returned by the **pm_initialize** subroutine.

The counting mode includes the User Mode and the Kernel Mode, or either of them; the Initial Counting State; and the Process Tree Mode. The Process Tree Mode sets counting to On only for the calling process and its descendants. The defaults are set to Off for the User Mode and the Kernel Mode. The initial default state is set to delay counting until the **pm_start** subroutine is called, and to count the activity of all the processes running in the system.

When you use the `pm_set_program_mm` subroutine for multi-mode counting, the Process Tree Mode and the Start Counting Mode are fixed by their values that are defined in the first programming set.

If the list includes an event that can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

On some platforms, event groups can be specified instead of individual events. This is done by setting the **is_group** bitfield in the mode, and placing the group ID into the first element of each events array. (The group ID was obtained by **pm_init** subroutine.)

Parameters

Item	Description
**prog*	Specifies the events and modes to use in Performance Monitor setup. It supports the following modes: ***PM_USER*** Counts processes that run in the User Mode (default is set to Off). ***PM_KERNEL*** Counts processes that run in the Kernel Mode (default is set to Off). ***PM_COUNT*** Starts counting immediately (default is set to Not to Start Counting). ***PM_PROCTREE*** Sets counting to On only for the calling process and its descendants (default is set to Off).
**prog_mm*	Specifies the events and the associated modes to use in the Performance Monitor setup. It supports the following modes: ***PM_USER*** Counts processes that run in the User Mode (default is set to Off). ***PM_KERNEL*** Counts processes that run in the Kernel Mode (default is set to Off). ***PM_COUNT*** Starts counting immediately (default is set to Not to start counting). ***PM_PROCTREE*** Sets counting to On only for the calling process and its descendants (default is set to Off). The *PM_PROCTREE* and the *PM_COUNT* modes defined in the first setting fix the value for the counting.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_mygroup Subroutine

Purpose

Sets Performance Monitor programmation for the calling thread and creates a counting group.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_mygroup ( *prog)
pm_prog_t *prog;
```

Description

The **pm_set_program_mygroup** subroutine sets the Performance Monitor programmation for the calling kernel thread. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the **pm_init** subroutine.

This call also creates a counting group, which includes the calling thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the calling process.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the inital default state is set to delay counting until the **pm_start_mygroup** subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

Parameters

Item	Description
**prog*	Specifies the events and mode to use in Performance Monitor setup. The following modes are supported: **PM_USER** Counts processes running in User Mode (default is set to Off) **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off) **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting) **PM_PROCESS** Creates a process-level counting group

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_mygroup_mx and pm_set_program_mygroup_mm Subroutines

Purpose

Sets Performance Monitor programmation in counter multiplexing mode and multi-mode for the calling thread and creates a counting group.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_mygroup_mx ( *prog)
pm_prog_mx_t *prog;
```

```
int pm_set_program_mygroup_mm (*prog_mm)
pm_prog_mm_t *prog_mm;
```

Description

The **pm_set_program_mygroup_mx** and pm_set_program_mygroup_mmsubroutines set the Performance Monitor programmation respectively in counter multiplexing mode or in multi-mode for the calling kernel thread.

The pm_set_program_mygroup_mx subroutine setting includes the list of event arrays to be counted and a mode in which to count. The mode is global to all of the event lists. The events to count are in an array of list of event identifiers.

The pm_set_program_mygroup_mm subroutine setting includes the list of the event arrays to be counted, and the mode in which to count each event array. A counting mode is defined for each event array.

The identifiers must be selected from the lists returned by the **pm_initialize** subroutine.

Both subroutines create a counting group, which includes the calling thread and any thread which it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future threads belonging to the calling process.

The counting mode for both subroutines includes the User Mode or the Kernel Mode, or both of them; the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the **pm_start_mygroup** subroutine is called.

When you use the pm_set_program_mygroup_mm subroutine for multi-mode counting, the Process Tree Mode and the Start Counting Mode are fixed by their values defined in the first programming set.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

Parameters

Item	Description
**prog*	Specifies the events and modes to use in Performance Monitor setup. The *prog* parameter supports the following modes: **PM_USER** Counts processes running in User Mode (default is set to Off). **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting). **PM_PROCESS** Creates a process-level counting group.

Item	Description
**prog_mm*	Specifies the events and the associated modes to use in the Performance Monitor setup. The *prog_mm* parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to start counting). **PM_PROCTREE** Sets counting to On only for the calling process and its descendants (default is set to Off). The *PM_PROCTREE* mode and the *PM_COUNT* mode defined in the first setting fix the value for the counting.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_mythread Subroutine

Purpose

Sets Performance Monitor programmation for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_mythread ( *prog)
pm_prog_t *prog;
```

Description

The **pm_set_program_mythread** subroutine sets the Performance Monitor programmation for the calling kernel thread. The setting includes the events to be counted, and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the **pm_init** subroutine.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the **pm_start_mythread** subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

Parameters

Item	Description
**prog*	Specifies the event modes to use in Performance Monitor setup. The following modes are supported: **PM_USER** Counts processes running in User Mode (default is set to Off) **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off) **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting) **PM_PROCESS** Creates a process-level counting group

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_mythread_mx and pm_set_program_mythread_mm Subroutines

Purpose

Sets Performance Monitor programmation in counter multiplexing mode and multi-mode for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_mythread_mx ( *prog)
pm_prog_mx_t *prog;

int pm_set_program_mythread_mm ( *prog_mm)
pm_prog_mm_t *prog_mm;
```

Description

The **pm_set_program_mythread_mx** and the `pm_set_program_mythread_mm` subroutines set the Performance Monitor programmation respectively in counter multiplexing mode or in multi-mode for the calling kernel thread.

The `pm_set_program_mythread_mx` subroutine setting includes the list of the event arrays to be counted, and a mode in which to count. The mode is global to all event lists. The events to count are in an array of list of event identifiers.

The `pm_set_program_mythread_mm` setting includes the lists of the event arrays to be counted, and the associated modes in which to count each event array. A counting mode is defined for each event array.

The event identifiers must be selected from the lists returned by the **pm_initialize** subroutine.

The counting mode for both subroutines includes the User Mode or the Kernel Mode, or both of them; and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the **pm_start_mythread** subroutine is called.

When you use the `pm_set_program_mythread_mm` subroutine for multi-mode counting, the Process Tree Mode and the Start Counting Mode are fixed by the their values defined in the first programming set.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

Parameters

Item	Description
**prog*	Specifies the events and the modes to use in the Performance Monitor setup. The *prog* parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting). **PM_PROCESS** Creates a process-level counting group.
**prog_mm*	Specifies the events and the modes to use in the Performance Monitor setup. The *prog_mm* parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to start counting). **PM_PROCTREE** Sets counting to On only for the calling process and its descendants (default is set to Off). The *PM_PROCTREE* mode and the *PM_COUNT* mode defined in the first setting fix the value for the counting.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_pgroup Subroutine

Purpose

Sets Performance Monitor programmation for a target pthread and creates a counting group.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_pgroup ( pid,  tid,  ptid,  *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_t *prog;
```

Description

The **pm_set_program_pgroup** subroutine sets the Performance Monitor programmation for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the **pm_inititialize** subroutine.

This call also creates a counting group, which includes the target pthread and any pthread that it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future pthreads belonging to the target process.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the initial default state is set to delay counting until the **pm_start_pgroup** subroutine is called.

If the list includes an event that can be used with a threshold (as indicated by the **pm_initialize** subroutine), a threshold value can also be specified.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Item	Description
**prog*	Specifies the event modes to use in Performance Monitor setup. The following modes are supported: **PM_USER** Counts processes running in User Mode (default is set to Off) **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off) **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting) **PM_PROCESS** Creates a process-level counting group

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_pgroup_mx and pm_set_program_pgroup_mm Subroutines

Purpose

Sets Performance Monitor programmation in counter multiplexing mode and multi-mode for a target pthread and creates a counting group.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_pgroup_mx ( pid,  tid,  ptid,  *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
```

```
pm_prog_mx_t *prog;

int pm_set_program_pgroup_mm ( pid, tid, ptid, *prog_mm)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_mm_t *prog_mm;
```

Description

The **pm_set_program_pgroup_mx** and the pm_set_program_pgroup_mm subroutines set the Performance Monitor programmation respectively in counter multiplexing mode or in multi-mode for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process.

The pm_set_program_pgroup_mx setting includes the list of the event arrays to be counted and a mode in which to count. The mode is global to all of the event lists. The events to count are in an array of list of event identifiers.

The pm_set_program_pgroup_mm setting includes the lists of the event arrays to be counted and the associated mode in which to count each event array. A counting mode is defined for each event array.

The event identifiers must be selected from the lists returned by the **pm_inititialize** subroutine.

Both subroutines create a counting group, which includes the target pthread and any pthread that it, or any of its descendants, will create in the future. Optionally, the group can be defined as also containing all the existing and future pthreads belonging to the target process.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The counting mode for both subroutines includes the User Mode, or the Kernel Mode, or both of them; and the Initial Counting State. The defaults are set to Off for the User Mode and the Kernel Mode, and the initial default state is set to delay counting until the **pm_start_pgroup** subroutine is called.

When you use the pm_set_program_pgroup_mm subroutine for multi-mode counting, the Process Tree Mode and the Start Counting Mode are fixed by their values defined in the first programming set.

If the list includes an event that can be used with a threshold (as indicated by the **pm_initialize** subroutine), a threshold value can also be specified.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Item	Description
**prog*	Specifies the events and the modes to use in the Performance Monitor setup. The *prog* parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting). **PM_PROCESS** Creates a process-level counting group.
**prog_mm*	Specifies the events and the modes to use in the Performance Monitor setup. The *prog_mm* parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to start counting). **PM_PROCTREE** Sets counting to On only for the calling process and its descendants (default is set to Off). The *PM_PROCTREE* mode and the *PM_COUNT* mode defined in the first setting fix the value for the counting.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_pthread Subroutine

Purpose

Sets Performance Monitor programmation for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_pthread ( pid,  tid,  ptid,  *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_t *prog;
```

Description

The **pm_set_program_pthread** subroutine sets the Performance Monitor programmation for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the **pm_inititialize** subroutine.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the Initial Default State is set to delay counting until the **pm_start_pthread** subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the **pm_initialize** subroutine), a threshold value can also be specified.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Item	Description
**prog*	Specifies the event modes to use in Performance Monitor setup. The following modes are supported: **PM_USER** Counts processes running in User Mode (default is set to Off) **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off) **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting)

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_pthread_mx and pm_set_program_pthread_mm Subroutines

Purpose

Sets Performance Monitor programmation in counter multiplexing mode and multi-mode for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_pthread_mx ( pid,  tid,  ptid,  *prog)
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_mx_t *prog;

int pm_set_program_pthread_mm ( pid, tid, ptid, *prog_mm)
```

```
pid_t pid;
tid_t tid;
ptid_t ptid;
pm_prog_mm_t *prog_mm;
```

Description

The **pm_set_program_pthread_mx** and the pm_set_program_pthread_mm subroutines set the Performance Monitor programmation respectively in counter multiplexing mode or in multi-mode for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process.

The pm_set_program_pthread_mx setting includes the list of the event arrays events to be counted and a mode in which to count. The mode is global to all of the event lists. The events to count are in an array of list of event identifiers.

The pm_set_program_pthread_mm subroutine setting includes the list of the event arrays to be counted, and the associated mode in which to count each event array. A counting mode is defined for each event array.

The event identifiers must be selected from the lists returned by the **pm_inititialize** subroutine.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

The counting mode for both subroutines includes the User Mode or the Kernel Mode, or both; and the Initial Counting State. The defaults are set to Off for the User Mode and the Kernel Mode, and the Initial Default State is set to delay counting until the **pm_start_pthread** subroutine is called.

When you use the pm_set_program_pthread_mm subroutine for multi-mode counting, the Process Tree Mode and the Start Counting Mode are fixed by their values defined in the first programming set.

If the list includes an event which can be used with a threshold (as indicated by the **pm_initialize** subroutine), a threshold value can also be specified.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.

Item	Description
**prog*	Specifies the events and the modes to use in the Performance Monitor setup. The *prog* parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting).
**prog_mm*	Specifies the events and the associated modes to use in the Performance Monitor setup. The *prog_mm* parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to start counting). **PM_PROCTREE** Sets counting to On only for the calling process and its descendants (default is set to Off). The *PM_PROCTREE* mode and the *PM_COUNT* mode defined in the first setting fix the value for the counting.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_thread Subroutine

Purpose

Sets Performance Monitor programmation for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_thread ( pid,  tid,  *prog)
pid_t pid;
tid_t tid;
pm_prog_t *prog;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_set_program_pthread** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_set_program_pthread** subroutine with a *ptid* parameter equal to 0.

The **pm_set_program_thread** subroutine sets the Performance Monitor programmation for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. The setting includes the events to be counted and a mode in which to count. The events to count are in a list of event identifiers. The identifiers must be selected from the lists returned by the **pm_init** subroutine.

The counting mode includes User Mode and/or Kernel Mode, and the Initial Counting State. The defaults are set to Off for User Mode and Kernel Mode, and the Initial Default State is set to delay counting until the **pm_start_thread** subroutine is called.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

Parameters

Item	Description
pid	Process ID of target thread. Target process must be a debuggee of the caller process.
tid	Thread ID of target thread.

Item	Description
**prog*	Specifies the event modes to use in Performance Monitor setup. The following modes are supported: **PM_USER** Counts processes running in User Mode (default is set to Off) **PM_KERNEL** Counts processes running in Kernel Mode (default is set to Off) **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting)

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_thread_mx and pm_set_program_thread_mm Subroutines

Purpose

Sets Performance Monitor programmation in counter multiplexing mode and multi-mode for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_set_program_thread_mx ( pid,  tid,  *prog)
pid_t pid;
tid_t tid;
pm_prog_mx_t *prog;

int pm_set_program_thread_mm ( pid, tid, *prog_mm)
pid_t pid;
tid_t tid;
pm_prog_mm_t *prog_mm;
```

Description

The pm_set_program_thread_mx and the pm_set_program_thread_mm subroutines support only the 1:1 threading model. They have been superseded respectively by the **pm_set_program_pthread_mx** and the pm_set_program_pthread_mm subroutines, which support both the 1:1 and the M:N threading models. A call to the pm_set_program_thread_mx subroutine or the pm_set_program_thread_mm subroutine is respectively equivalent to a call to the **pm_set_program_pthread_mx** subroutine or the pm_set_program_pthread_mm subroutine with a *ptid* parameter equal to 0.

The **pm_set_program_thread_mx** and the pm_set_program_thread_mm subroutines set the Performance Monitor programmation respectively in counter multiplexing mode or multi-mode for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process.

The pm_set_program_thread_mx setting includes the list of the event arrays to be counted and a mode in which to count. The mode is global to all of the event lists. The events to count are in an array of list of event identifiers.

The pm_set_program_thread_mm setting includes the list of the event arrays to be counted, and the associated mode in which to count each event array. A counting mode is defined for each event array.

The event identifiers must be selected from the lists returned by the **pm_initialize** subroutine.

The counting mode for both subroutines includes the User Mode, or the Kernel Mode, or both of them; and the Initial Counting State. The defaults are set to Off for the User Mode and the Kernel Mode, and the Initial Default State is set to delay counting until the **pm_start_thread** subroutine is called.

When you use the pm_set_program_thread_mm subroutine for the multi-mode counting, the Process Tree Mode and the Start Counting Mode are fixed by their values in the first programming set.

If the list includes an event which can be used with a threshold (as indicated by the **pm_init** subroutine), a threshold value can also be specified.

Parameters

Item	Description
pid	Process ID of target thread. Target process must be a debuggee of the caller process.
tid	Thread ID of target thread.
**prog*	Specifies the events and the modes to use in the Performance Monitor setup. The *prog* parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to Start Counting).

Item	Description
**prog_mm*	Specifies the events and the associated modes to use in the Performance Monitor setup. The prog_mm parameter supports the following modes: **PM_USER** Counts processes running in the User Mode (default is set to Off). **PM_KERNEL** Counts processes running in the Kernel Mode (default is set to Off). **PM_COUNT** Starts counting immediately (default is set to Not to start counting). **PM_PROCTREE** Sets counting to On only for the calling process and its descendants (default is set to Off). The *PM_PROCTREE* mode and the *PM_COUNT* mode defined in the first setting fix the value for the counting.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_wp Subroutine

Purpose

Sets Performance Monitor programming for a specified workload partition (WPAR).

Syntax

```
#include <pmapi.h>
int pm_set_program_wp (cid, *prog)
cid_t cid;
pm_prog_t *prog;
```

Description

The **pm_set_program_wp** subroutine sets Performance Monitor programming for the processes that belong to the specified workload partition (WPAR). The programming includes the events to be counted, and a mode in which to count.

The events to count are in a list of event identifiers. The identifiers must be selected from the list that the **pm_initialize** subroutine returns. If the list includes an event that can be used with a threshold, you can specify a threshold value.

In some platforms, you can specify an event group instead of individual events. Set the **is_group** bit field in the mode and type the group ID in the first element of the event array. The group ID can be obtained by the **pm_initialize** subroutine.

The counting mode includes both User mode and Kernel mode, or either of them; the Initial Counting state; and the Process Tree mode. If the Process Tree mode is set to the On state, the counting only applies to the calling process and its descendants. The default values for User mode and Kernel mode are `Off`. The initial default state is set to delay the counting until calling the **pm_start** subroutine, and to count the activities of all of the processes running into the specified WPAR.

Parameters

Item	Description
cid	Specifies the identifier of the WPAR for which the subroutine is to be set. The CID can be obtained from the WPAR name using the **getcorralid** system call.
prog	Specifies the events and modes to use in Performance Monitor setup. The following modes are supported: **PM_USER** Counts processes that are running in User mode. The default value is set to `Off`. **PM_KERNEL** Counts processes that are running in Kernel mode. The default value is set to `Off`. **PM_COUNT** Starts counting immediately. The default value is set to `Not to start counting`. **PM_PROCTREE** Sets counting to On for only the calling process and its descendants. The default value is set to `Off`.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Run the **pm_error** subroutine to decode the error code.

Error codes

To decode the error code, see the **pm_error** subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_set_program_wp_mm Subroutine

Purpose

Sets Performance Monitor programming in counter multiplexing mode for a specified workload partition.

Syntax

```
#include <pmapi.h>

int pm_set_program_wp_mm (cid, *prog_mm)
cid_t cid;
pm_prog_mm_t *prog_mm;
```

Description

The **pm_set_program_wp_mm** subroutine sets Performance Monitor programming in counter multiplexing mode for the processes that belong to a specified workload partition (WPAR). The programming includes the list of the event arrays to be counted, and the associated mode in which to count each event array. A counting mode is defined for each event array. The identifiers must be selected from the lists that the “pm_initialize Subroutine” on page 1302 subroutine returns. If the list includes an event that can be used with a threshold, you can specify a threshold value.

In some platforms, you can specify an event group instead of individual events. Set the **is_group** bit field in the mode and type the group ID in the first element of each event array. The group ID can be obtained by the **pm_initialize** subroutine.

The counting mode includes both User mode and Kernel mode, or either of them; the Initial Counting state; and the Process Tree mode. The default values for User mode and Kernel mode are `Off`. The initial default state is set to delay the counting until calling the **pm_start** subroutine (“pm_start and pm_tstart Subroutine” on page 1341), and to count the activities of all of the processes running into the specified WPAR.

If you use the **pm_set_program_wp_mm** subroutine for a multi-mode counting, Process Tree mode (PM_PROCTREE) and Start Counting mode (PM_COUNT) retain the values that are defined in the first programming set.

If the Process Tree mode is set to the On state, the counting only applies to the calling process and its descendants.

Parameters

Item	Description
cid	Specifies the identifier of the WPAR for which the programming is to be set. The CID can be obtained from the WPAR name using the **getcorralid** system call.

Item	Description
prog_mm	Specifies the events and associated modes to use in Performance Monitor setup. The following modes are supported: **PM_USER** Counts processes that are running in User mode. The default value is set to `Off`. **PM_KERNEL** Counts processes that are running in Kernel mode. The default value is set to `Off`. **PM_COUNT** Starts counting immediately. The default value is set to `Not to start counting`. **PM_PROCTREE** Sets counting to On for only the calling process and its descendants. The default value is set to `Off`.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Run the **pm_error** subroutine (“pm_error Subroutine” on page 1246) to decode the error code.

Error Codes

To decode the error code, see the **pm_error** subroutine (“pm_error Subroutine” on page 1246).

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_start and pm_tstart Subroutine

Purpose

Starts system wide Performance Monitor counting.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_start()

int pm_tstart(*time)
timebasestruct_t *time;
```

Description

The **pm_start** subroutine starts system wide Performance Monitor counting.

The **pm_tstart** subroutine starts system wide Performance Monitor counting, and returns a timestamp indicating when the counting was started.

Parameters

Item	Description
**time*	Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the "pm_error Subroutine" on page 1246 to decode the error code

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_start_group and pm_tstart_group Subroutine

Purpose

Starts Performance Monitor counting for the counting group to which a target thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_start_group ( pid,  tid)
pid_t pid;
tid_t tid;

int pm_tstart_group ( pid,  tid, *time)
pid_t pid;
tid_t tid;
timebasestruct_t *time
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_start_pgroup** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_start_pgroup** subroutine with a *ptid* parameter equal to 0.

The **pm_start_group** subroutine starts the Performance Monitor counting for a target kernel thread and the counting group to which it belongs. This counting is effective immediately for the target thread. For all the other thread members of the counting group, the counting will start after their next redispatch, but only if their current counting state is already set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state. If their counting state is currently set to Off, no counting starts until they call either the **pm_start_mythread** subroutine or the **pm_start_mygroup** themselves, or until a debugger process calls the **pm_start_thread** subroutine or the **pm_start_group** subroutine on their behalf.

The **pm_tstart_group** subroutine starts the Performance Monitor counting for a target kernel thread and the counting group to which it belongs, and returns a timestamp indicating when the counting was started.

Parameters

Item	Description
pid	Process ID of target thread. Target process must be a debuggee of the caller process.
tid	Thread ID of target thread.
**time*	Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_start_mygroup and pm_tstart_mygroup Subroutine

Purpose

Starts Performance Monitor counting for the group to which the calling thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int_pm_start_mygroup()

int pm_tstart_mygroup (*time)
timebasestruct_t *time
```

Description

The **pm_start_mygroup** subroutine starts the Performance Monitor counting for the calling kernel thread and the counting group to which it belongs. Counting is effective immediately for the calling thread. For all the other threads members of the counting group, the counting starts after their next redispatch, but only if their current counting state is already set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state. If their counting state is currently set to Off, no counting starts until they call either the **pm_start_mythread** subroutine or the **pm_start_mygroup** subroutine themselves, or until a debugger process calls the **pm_start_thread** subroutine or the **pm_start_group** subroutine on their behalf.

The **pm_tstart_mygroup** subroutine starts the Performance Monitor counting for the calling kernel thread and the counting group to which it belongs, and returns a timestamp indicating when the counting was started.

Parameters

Item	Description
**time*	Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_start_mythread and pm_tstart_mythread Subroutine

Purpose

Starts Performance Monitor counting for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_start_mythread()

int pm_tstart_mythread(*time)
timebasestruct_t *time;
```

Description

The **pm_start_mythread** subroutine starts Performance Monitor counting for the calling kernel thread. Counting is effective immediately unless the thread is in a group, and that group's counting is not currently set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state.

The **pm_tstart_mythread** subroutine starts Performance Monitor counting for the calling kernel thread, and returns a timestamp indicating when the counting was started.

Parameters

Item	Description
**time*	Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_start_pgroup and pm_tstart_pgroup Subroutine

Purpose

Starts Performance Monitor counting for the counting group to which a target pthread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_start_pgroup ( pid,  tid,  ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;

int pm_tstart_pgroup ( pid,  tid, ptid, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time
```

Description

The **pm_start_pgroup** subroutine starts the Performance Monitor counting for a target pthread and the counting group to which it belongs. This counting is effective immediately for the target pthread. For all the other thread members of the counting group, the counting will start after their next redispatch, but only if their current counting state is already set to On. The counting state of a pthread in a group is obtained by ANDing the pthread counting state with the group state. If their counting state is currently set to Off, no counting starts until they call either the **pm_start_mythread** subroutine or the **pm_start_mygroup** themselves, or until a debugger process calls the **pm_start_pthread** subroutine or the **pm_start_pgroup** subroutine on their behalf.

The **pm_tstart_pgroup** subroutine starts the Performance Monitor counting for a target pthread and the counting group to which it belongs, and returns a timestamp indicating when the counting was started.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**time*	Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_start_pthread and pm_tstart_pthread Subroutine

Purpose

Starts Performance Monitor counting for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_start_pthread ( pid,  tid,  ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;

int pm_start_pthread ( pid,  tid,  ptid, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time
```

Description

The **pm_start_pthread** subroutine starts Performance Monitor counting for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process. Counting is effective immediately unless the thread is in a group and the group counting is not currently set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state.

The **pm_tstart_pthread** subroutine starts Performance Monitor counting for a target pthread, and returns a timestamp indicating when the counting was started.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.

Item	Description
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**time*	Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the "pm_error Subroutine" on page 1246 to decode the error code.

Error Codes

Refer to the "pm_error Subroutine" on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_start_thread and pm_tstart_thread Subroutine

Purpose

Starts Performance Monitor counting for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_start_thread (pid, tid)
pid_t pid;
tid_t tid;

int pm_tstart_thread ( pid,  tid, *time)
pid_t pid;
tid_t tid;
timebasestruct_t *time
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_start_pthread** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_start_pthread** subroutine with a *ptid* parameter equal to 0.

The **pm_start_thread** subroutine starts Performance Monitor counting for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process. Counting is effective immediately unless the thread is in a group and the group counting is not currently set to On. The counting state of a thread in a group is obtained by ANDing the thread counting state with the group state.

The **pm_tstart_thread** subroutine starts Performance Monitor counting for a target kernel thread, and returns a timestamp indicating when the counting was started.

Parameters

Item	Description
pid	Process ID of target thread. Target process must be a debuggee of the caller process.
tid	Thread ID of target thread.
**time*	Pointer to a structure containing the timebase value when the counting was started. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	Operation completed successfully.
Positive Error Code	Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine to decode the error code.

Error Codes

Refer to the **pm_error** (“pm_error Subroutine” on page 1246) subroutine.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_start_wp and pm_tstart_wp Subroutines

Purpose

Starts Performance Monitor counting for a specified workload partition.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_start_wp(cid)
cid_t cid;

int pm_tstart_wp(cid, *time)
cid_t cid;
timebasestruct_t *time;
```

Description

The **pm_start_wp** and **pm_tstart_wp** subroutines start counting for the activities of the processes that belong to a specified workload partition (WPAR).

The **pm_start_wp** subroutine starts Performance Monitor counting for a specified WPAR.

The **pm_tstart_wp** subroutine starts Performance Monitor counting for a specified WPAR, and returns a timestamp indicating when the counting was started.

Parameters

Item	Description
cid	Specifies the WPAR identifier that the counting starts from. The CID can be obtained from the WPAR name using the **getcorralid** system call.
time	Pointer to a structure that contains the *timebase* value when the counting starts. The value of *time* can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Run the **pm_error** subroutine (“pm_error Subroutine” on page 1246) to decode the error code.

Error Codes

Run the **pm_error** subroutine to decode the error code.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_stop and pm_tstop Subroutine

Purpose

Stops system wide Performance Monitor counting.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_stop ()

int pm_tstop(*time)
timebasestruct_t *time;
```

Description

The **pm_stop** subroutine stops system wide Performance Monitoring counting.

The **pm_tstop** subroutine stops system wide Performance Monitoring counting, and returns a timestamp indicating when the counting was stopped.

Parameters

Item	Description
**time*	Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_stop_group and pm_tstop_group Subroutine

Purpose

Stops Performance Monitor counting for the group to which a target thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_stop_group ( pid,  tid)
pid_t pid;
tid_t tid;
```

```
int pm_tstop_group ( pid,  tid, *time )
pid_t pid;
tid_t tid;
timebasestruct_t *time;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_stop_pgroup** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_stop_pgroup** subroutine with a *ptid* parameter equal to 0.

The **pm_stop_group** subroutine stops Performance Monitor counting for a target kernel thread, the counting group to which it belongs, and all the other thread members of the same group. Counting stops immediately for all the threads in the counting group. The target thread must be stopped and must be part of a debuggee process, under control of the calling process.

The **pm_tstop_group** subroutine stops Performance Monitor counting for a target kernel thread, the counting group to which it belongs, and all the other thread members of the same group, and returns a timestamp indicating when the counting was stopped.

Parameters

Item	Description
pid	Process ID of target thread. Target process must be a debuggee of the caller process.
tid	Thread ID of target thread.
**time*	Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_stop_mygroup and pm_tstop_mygroup Subroutine

Purpose

Stops Performance Monitor counting for the group to which the calling thread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_stop_mygroup ()

int pm_tstop_mygroup(*time)
timebasestruct_t *time;
```

Description

The **pm_stop_mygroup** subroutine stops Performance Monitor counting for the group to which the calling kernel thread belongs. This is effective immediately for all the threads in the counting group.

The **pm_tstop_mygroup** subroutine stops Performance Monitor counting for the group to which the calling kernel thread belongs, and returns a timestamp indicating when the counting was stopped.

Parameters

Item	Description
**time*	Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_stop_mythread and pm_tstop_mythread Subroutine

Purpose

Stops Performance Monitor counting for the calling thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_stop_mythread ()

int pm_tstop_mythread(*time)
timebasestruct_t *time;
```

Description

The **pm_stop_mythread** subroutine stops Performance Monitor counting for the calling kernel thread.

The **pm_tstop_mythread** subroutine stops Performance Monitor counting for the calling kernel thread, and returns a timestamp indicating when the counting was stopped.

Parameters

Item	Description
**time*	Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_stop_pgroup and pm_tstop_pgroup Subroutine

Purpose

Stops Performance Monitor counting for the group to which a target pthread belongs.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_stop_pgroup ( pid,  tid,  ptid)
pid_t pid;
tid_t tid;
```

```
ptid_t ptid;

int pm_tstop_pgroup ( pid,  tid,  ptid, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time;
```

Description

The **pm_stop_pgroup** subroutine stops Performance Monitor counting for a target pthread, the counting group to which it belongs, and all the other pthread members of the same group. Counting stops immediately for all the pthreads in the counting group. The target pthread must be stopped and must be part of a debuggee process, under control of the calling process.

The **pm_tstop_pgroup** subroutine stops Performance Monitor counting for a target pthread, the counting group to which it belongs, and all the other pthread members of the same group, and returns a timestamp indicating when the counting was stopped.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**time*	Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_stop_pthread and pm_tstop_pthread Subroutine

Purpose

Stops Performance Monitor counting for a target pthread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_stop_pthread ( pid,  tid,  ptid)
pid_t pid;
tid_t tid;
ptid_t ptid;

int pm_tstop_pthread ( pid,  tid,  ptid, *time)
pid_t pid;
tid_t tid;
ptid_t ptid;
timebasestruct_t *time;
```

Description

The **pm_stop_pthread** subroutine stops Performance Monitor counting for a target pthread. The pthread must be stopped and must be part of a debuggee process, under the control of the calling process.

The **pm_tstop_pthread** subroutine stops Performance Monitor counting for a target pthread, and returns a timestamp indicating when the counting was stopped.

If the pthread is running in 1:1 mode, only the *tid* parameter must be specified. If the pthread is running in m:n mode, only the *ptid* parameter must be specified. If both the *ptid* and *tid* parameters are specified, they must be referring to a single pthread with the *ptid* parameter specified and currently running on a kernel thread with specified *tid* parameter.

Parameters

Item	Description
pid	Process ID of target pthread. Target process must be a debuggee of the caller process.
tid	Thread ID of target pthread. To ignore this parameter, set it to 0.
ptid	Pthread ID of the target pthread. To ignore this parameter, set it to 0.
**time*	Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	Operation completed successfully.

Item	Description
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_stop_thread and pm_tstop_thread Subroutine

Purpose

Stops Performance Monitor counting for a target thread.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_stop_thread (pid, tid)
pid_t pid;
tid_t tid;

int pm_tstop_thread (pid,  tid, *time)
pid_t pid;
tid_t tid;
timebasestruct_t *time;
```

Description

This subroutine supports only the 1:1 threading model. It has been superseded by the **pm_stop_pthread** subroutine, which supports both the 1:1 and the M:N threading models. A call to this subroutine is equivalent to a call to the **pm_stop_pthread** subroutine with a *ptid* parameter equal to 0.

The **pm_stop_thread** subroutine stops Performance Monitor counting for a target kernel thread. The thread must be stopped and must be part of a debuggee process, under the control of the calling process.

The **pm_tstop_thread** subroutine stops Performance Monitor counting for a target kernel thread, and returns a timestamp indicating when the counting was stopped.

Parameters

Item	Description
pid	Process ID of target thread. Target process must be a debuggee of the caller process.
tid	Thread ID of target thread.

Item	Description
**time*	Pointer to a structure containing the timebase value when the counting was stopped. This can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
`0`	Operation completed successfully.
`Positive error code`	Refer to the “pm_error Subroutine” on page 1246 to decode the error code.

Error Codes

Refer to the “pm_error Subroutine” on page 1246.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pm_stop_wp and pm_tstop_wp Subroutines

Purpose

Stops Performance Monitor counting for a specified workload partition.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>

int pm_stop_wp (cid)
cid_t cid;

int pm_tstop_wp(cid, *time)
cid_t cid;
timebasestruct_t *time;
```

Description

The **pm_stop_wp** and **pm_tstop_wp** subroutines stop counting for the activities of the processes that belong to a specified workload partition (WPAR).

The **pm_stop_wp** subroutine stops Performance Monitor counting for a specified WPAR.

The **pm_tstop_wp** subroutine stops Performance Monitor counting for a specified WPAR, and returns a timestamp indicating when the counting was started.

Parameters

Item	Description
cid	Specifies the WPAR identifier from which the counting stops. The CID can be obtained from the WPAR name using the **getcorralid** system call.
time	Pointer to a structure that contains the *timebase* value when the counting starts. The value of *time* can be converted to time using the **time_base_to_time** subroutine.

Return Values

Item	Description
0	Operation completed successfully.
Positive error code	Run the **pm_error** subroutine ("pm_error Subroutine" on page 1246) to decode the error code.

Error Codes

Run the **pm_error** subroutine to decode the error code.

Files

Item	Description
/usr/include/pmapi.h	Defines standard macros, data types, and subroutines.

pmc_read_1to4 Subroutine

Purpose

Reads the performance monitoring counters (PMC) registers PMC 1 to PMC 4 in problem state.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int pmc_read_1to4 (void * buffer)
```

Description

The **pmc_read_1to4** subroutine reads the registers PMC 1 to PMC 4 into the address of the buffer that is passed as a parameter to the function.

All the four 32- bit PMC registers such as **PMC1**, **PMC2**, **PMC3**, and **PMC4** are read in the same order into the buffer.

Return Values

If the read operation is successful, a value of zero is returned. If the read operation fails, a value of -1 is returned.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

pmc_read_5to6 Subroutine

Purpose

Reads the performance monitoring counter (PMC) 5 and 6 (PMC5 and PMC6) in the problem state.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int pmc_read_5to6 (void * buffer)
```

Description

The **pmc_read_5to6** subroutine reads the registers PMC 5 and PMC 6 in the same order into the address of the buffer that is passed as a parameter to the function.

The two 32-bit PMC registers (**PMC5** and **PMC6**) are read into the buffer.

Return Values

If the read operation is successful, a value of zero is returned. If the read operation fails, a value of -1 is returned.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

pmc_write Subroutine

Purpose

Writes a performance monitor control (PMC) in problem state.

Library

Performance Monitor APIs Library (**libpmapi.a**)

Syntax

```
#include <pmapi.h>
int _write (int reg_num, void *buffer)
```

Description

The **pmc_write** subroutine writes a PMC in problem state.

The function takes two parameters namely the Special Purpose Register (SPR) number of the PMC register into which the value is written and the address from where the value is written to the PMC SPR.

The **pmc_write** subroutine writes the value of the address specified in the second argument into the register specified in the first argument.

Return Values

If the write operation is successful, a value of zero is returned. If the write operation fails, a value of -1 is returned.

Files

The **pmapi.h** file defines standard macros, data types, and subroutines.

poll Subroutine

Purpose

Checks the I/O status of multiple file descriptors and message queues.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/poll.h>
#include <sys/select.h>
#include <sys/types.h>
```

```
int poll( ListPointer,  Nfdsmsgs,  Timeout)
void *ListPointer;
unsigned long Nfdsmsgs;
long Timeout;
```

Description

The **poll** subroutine checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or to see if they have an exceptional condition pending. Even though there are **OPEN_MAX** number of file descriptors available, only **FD_SETSIZE** number of file descriptors are accessible with this subroutine.

Note: The **poll** subroutine applies only to character devices, pipes, message queues, and sockets. Not all character device drivers support it. See the descriptions of individual character devices for information about whether and how specific device drivers support the **poll** and **select** subroutines.

For compatibility with previous releases of this operating system and with BSD systems, the **select** subroutine is also supported.

If a program needs to use message queue support, the program source code should be compiled with the **-D_MSGQSUPPORT** compilation flag.

Parameters

Item	Description

ListPointer	Specifies a pointer to an array of **pollfd** structures, **pollmsg** structures, or to a**pollist** structure. Each structure specifies a file descriptor or message queue ID and the events of interest for this file or message queue. The **pollfd**, **pollmsg**, and **pollist** structures are defined in the **/usr/include/sys/poll.h** file. If a **pollist** structure is to be used, a structure similar to the following should be defined in a user program. The **pollfd** structure must precede the **pollmsg** structure.
	`struct pollist {`
	`  struct pollfd fds[3];`
	`  struct pollmsg msgs[2];`
	`  } list;`
	The structure can then be initialized as follows:
	`list.fds[0].fd = file_descriptorA;`
	`list.fds[0].events = requested_events;`
	`list.msgs[0].msgid = message_id;`
	`list.msgs[0].events = requested_events;`
	The rest of the elements in the**fds**and**msgs**arrays can be initialized the same way. The **poll** subroutine can then be called, as follows:
	`nfds = 3;  /* number of pollfd structs */`
	`nmsgs = 2;  /* number of pollmsg structs */`
	`timeout = 1000  /* number of milliseconds to timeout */`
	`poll(&list, (nmsgs<<16)|(nfds), 1000);`
	The exact number of elements in the **fds** and **msgs** arrays must be used in the calculation of the *Nfdsmsgs* parameter.

Nfdsmsgs	Specifies the number of file descriptors and the exact number of message queues to check. The low-order 16 bits give the number of elements in the array of **pollfd** structures, while the high-order 16 bits give the exact number of elements in the array of **pollmsg** structures. If either half of the*Nfdsmsgs* parameter is equal to a value of 0, the corresponding array is assumed not to be present.
Timeout	Specifies the maximum length of time (in milliseconds) to wait for at least one of the specified events to occur. If the *Timeout* parameter value is -1, the **poll** subroutine does not return until at least one of the specified events has occurred. If the value of the *Timeout* parameter is 0, the **poll** subroutine does not wait for an event to occur but returns immediately, even if none of the specified events have occurred.

poll Subroutine STREAMS Extensions

In addition to the functions described above, the **poll** subroutine multiplexes input/output over a set of file descriptors that reference open streams. The **poll** subroutine identifies those streams on which you can send or receive messages, or on which certain events occurred.

You can receive messages using the **read** subroutine or the **getmsg** system call. You can send messages using the **write** subroutine or the **putmsg** system call. Certain **streamio** operations, such as **I_RECVFD** and **I_SENDFD** can also be used to send and receive messages. See the **streamio** operations.

The *ListPointer* parameter specifies the file descriptors to be examined and the events of interest for each file descriptor. It points to an array having one element for each open file descriptor of interest. The array's elements are **pollfd** structures. In addition to the **pollfd** structure in the /**usr**/**include**/**sys**/**poll.h** file, STREAMS supports the following members:

```
int fd;             /*  file
descriptor  */ short events;       /* requested events */
short revents;      /* returned events  */
```

The `fd` field specifies an open file descriptor and the `events` and `revents` fields are bit-masks constructed by ORing any combination of the following event flags:

Item	Description
POLLIN	A nonpriority or file descriptor-passing message is present on the stream-head read queue. This flag is set even if the message is of 0 length. In the `revents` field this flag is mutually exclusive with the **POLLPRI** flag. See the **I_RECVFD** command.
POLLRDNORM	A nonpriority message is present on the stream-head read queue.
POLLRDBAND	A priority message (band > 0) is present on the stream-head read queue.
POLLPRI	A high-priority message is present on the stream-head read queue. This flag is set even if the message is of 0 length. In the `revents` field, this flag is mutually exclusive with the **POLLIN** flag.
POLLOUT	The first downstream write queue in the stream is not full. Normal priority messages can be sent at any time. See the **putmsg** system call.
POLLWRNORM	The same as **POLLOUT**.
POLLWRBAND	A priority band greater than 0 exists downstream and priority messages can be sent at anytime.
POLLMSG	A message containing the **SIGPOLL** signal has reached the front of the stream-head read queue.

Return Values

On successful completion, the **poll** subroutine returns a value that indicates the total number of file descriptors and message queues that satisfy the selection criteria. The return value is similar to the *Nfdsmsgs* parameter in that the low-order 16 bits give the number of file descriptors, and the high-order 16 bits give the number of message queue identifiers that had nonzero `revents` values. The **NFDS** and **NMSGS** macros, found in the **sys/select.h** file, can be used to separate these two values from the return value. The **NFDS** macro returns **NFDS#**, where the number returned indicates the number of files reporting some event or error, and the **NMSGS** macro returns **NMSGS#**, where the number returned indicates the number of message queues reporting some event or error.

A value of 0 indicates that the **poll** subroutine timed out and that none of the specified files or message queues indicated the presence of an event (all `revents` fields were values of 0).

If unsuccessful, a value of -1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **poll** subroutine does not run successfully if one or more of the following are true:

Item	Description
EAGAIN	Allocation of internal data structures was unsuccessful.
EINTR	A signal was caught during the **poll** system call and the signal handler was installed with an indication that subroutines are not to be restarted.
EINVAL	The number of **pollfd** structures specified by the *Nfdsmsgs* parameter is greater than **FD_SETSIZE**. This error is also returned if the number of **pollmsg** structures specified by the *Nfdsmsgs* parameter is greater than the maximum number of allowable message queues.
EFAULT	The *ListPointer* parameter in conjunction with the *Nfdsmsgs* parameter addresses a location outside of the allocated address space of the process.

pollset_create, pollset_ctl, pollset_destroy, pollset_poll, pollset_query, pollset_ctl_ext, pollset_poll_ext, pollset_query_ext, and pollset_ext Subroutines

Purpose

Check I/O status of multiple file descriptors.

Library

Standard C Library (`libc.a`)

Syntax

```
#include <sys/poll.h>
#include <sys/pollset.h>
#include <sys/fcntl.h>

pollset_t ps = pollset_create(int maxfd)
int rc = pollset_destroy(pollset_t ps)
int rc = pollset_ctl(pollset_t ps, struct poll_ctl *pollctl_array,
                     int array_length)
int rc = pollset_query(pollset_t ps, struct pollfd *pollfd_query)
int nfound = pollset_poll(pollset_t ps,
                                    struct pollfd *polldata_array,
                                    int array_length, int timeout)
int rc = pollset_ctl_ext(pollset_t ps, struct poll_ctl_ext* pollctl_array,
                         int array_length)
int rc = pollset_query_ext(pollset_t ps, struct pollfd_ext *pollfd_query)
int nfound = pollset_poll_ext(pollset_t ps,
                                    struct pollfd_ext *polldata_array,
                                    int array_length, int timeout)
int pollset_ext(void)
```

Description

The `pollset` application programming interface (API) efficiently poll a large file descriptor set. This interface is best used when the file descriptor set is not frequently updated. The `pollset` subroutine can provide a significant performance enhancement over traditional `select` and `poll` APIs. Improvements are most visible when the number of events returned per poll operation is small in relation to the number of file descriptors polled.

The `pollset` API uses system calls to accomplish polling. A file descriptor set (or *pollset*) is established with a successful call to `pollset_create`. File descriptors and poll events are added, removed, or updated using the `pollset_ctl` subroutine. The `pollset_poll` subroutine is called to perform the poll operation. A `pollset_query` subroutine is called to query if a file descriptor is contained in the current poll set. Extended versions of the control, poll, and query subroutines exist to ease consumption for applications that desire to use features only present with the extended *pollset* formats.

A pollset is established with a successful call to `pollset_create`. The pollset is initially empty following this system call. Each call to `pollset_create` creates a new and independent pollset. This can be useful to applications that monitor distinct sets of file descriptors. The maximum number of file descriptors that can belong to the pollset is specified by *maxfd*. If *maxfd* has a value of `-1`, the maximum number of file descriptors that can belong to the pollset is bound by `OPEN_MAX` as defined in `<sys/limits.h>` (the AIX limit of open file descriptors per process). AIX imposes a system-wide limit of 245025 active pollsets at one time. Upon failure, this system call returns `-1` with `errno` set appropriately. Upon success, a pollset ID of type `pollset_t` is returned:

```
        typedef int pollset_t
```

The pollset ID must not be altered by the application. The pollset API verifies that the ID is not `-1`. In addition, the process ID of the application must match the process ID stored at pollset creation time.

A pollset is destroyed with a successful call to pollset_destroy. Upon success, this system call returns 0. Upon failure, the pollset_destroy subroutine returns -1 with errno set to the appropriate code. An errno of EINVAL indicates an invalid pollset ID.

File descriptors must be added to the pollset with the pollset_ctl subroutine or pollset_ctl_ext subroutine before they can be monitored. A heterogeneous array of poll_ctl structures and/or poll_ctl_ext structures is passed to pollset_ctl or pollset_ctl_ext through pollctl_array. The **array_length** parameter indicates the number of distinct poll_ctl or poll_ctl_ext elements in the array.

Each poll_ctl poll control structure contains a version, *command*, *fd*, and *events* field. This structure is backwards compatible with the definition from previous releases of AIX when the version field is set to 0. An extended poll control structure, poll_ctl_ext, adds an additional field to store user-specified data when the version field is set to 1. The *fd* field defines the file descriptor that is the target of the command. The *events* field contains events of interest. When the is PS_ADD, the pollset_ctl or pollset_ctl_ext call adds a valid open file descriptor to the pollset. If a file descriptor is already in the pollset, PS_ADD causes pollset_ctl or pollset_ctl_ext to return an error. When the command is PS_MOD and the file descriptor is already in the pollset, bits in the *events* field are added (ORed) to the monitored events. If the file descriptor is not already in the pollset, the PS_MOD behavior is equivalent to PS_ADD, and adds a valid open file descriptor to the pollset.

Poll events and user-specified data can be refreshed for a file descriptor that is currently resident in the pollset with the **PS_REPLACE** command. When command is PS_DELETE and the file descriptor is already in the pollset, pollset_ctl or pollset_ctl_ext removes the file descriptor from the pollset. If the file descriptor is not already in the pollset, then PS_DELETE causes pollset_ctl or pollset_ctl_ext to return an error.

The pollset_query or pollset_query_ext interface can be used to determine information about a file descriptor with respect to the pollset. When the file descriptor is in the pollset, pollset_query or pollset_query_ext returns 1 and *events* is set to the currently monitored events. When present, user-specified data for the file descriptor is also returned and indicated by the presence of the POLLEXT event.

The pollset_poll or pollset_poll_ext subroutine determines which file descriptors in the pollset that have events pending. The *polldata_array* parameter contains a buffer address where pollfd or pollfd_ext structures are returned for file descriptors that have pending events. The number of events returned by a poll is limited by *array_length*. The *timeout* parameter specifies the amount of time to wait if no events are pending. Setting *timeout* to 0 guarantees that the pollset_poll or pollset_poll_ext subroutine returns immediately. Setting timeout to -1 specifies an infinite timeout. Other nonzero positive values specify the time to wait in milliseconds.

When events are returned from a pollset_poll or pollset_poll_ext operation, each pollfd or pollfd_ext structure contains an *fd* field with the file descriptor set, an *events* field with the requested events, and a *revents* field with the events that have occurred. Pollset events which contain user-specified data will return it in the data field of the pollfd_ext structure and indicate its presence with the POLLEXT event in the *events* field.

A single pollset can be accessed by multiple threads in a multithreaded process. When multiple threads are polling one pollset and an event occurs for a file descriptor, only one thread might be prompted to receive the event. After a file descriptor is returned to a thread, new events will not be generated until the next pollset_poll or pollset_poll_ext call. This behavior prevents all threads from being prompted on each event. Multiple threads can perform pollset_poll or control operations concurrently. If a user wishes to have serialized pollset modifications, environment variable *PS_CTL_BLOCKING* should be set to yes to have pollset_ctl or pollset_ctl_ext calls block until all running threads in pollset_poll or pollset_poll_ext have exited. A thread calling pollset_destroy is blocked until all threads have left the pollset poll, control, and query system calls.

A process can call fork after calling pollset_create. The child process will already have a pollset ID per pollset, but pollset_destroy, pollset_ctl, pollset_ctl_ext, pollset_query, pollset_query_ext, pollset_poll, and pollset_poll_ext operations will fail with an errno value of EACCES.

After a file descriptor is added to a pollset, the file descriptor will not be removed until a `pollset_ctl` or a `pollset_ctl_ext` call along with the PS_DELETE command is run. The file descriptor remains in the pollset even if the file descriptor is closed. A `pollset_poll` or `pollset_poll_ext` operation on a pollset containing a closed file descriptor returns a POLLNVAL event for that file descriptor. If the file descriptor is later allocated to a new object, the new object will be polled on future `pollset_poll` or `pollset_poll_ext` calls.

Applications may use `pollset_ext` to support operation in AIX versions that do not support extended pollset features. A return value of `true` (`1`) indicates extended pollsets are supported, while a return value of `false` (`0`) indicates that only the legacy pollset structures and APIs are supported.

Parameters

Item	Description
array_length	Specifies the length of the array parameters.
maxfd	Specifies the maximum number of file descriptors that can belong to the pollset.
pollctl_array	The pointer to a homogeneous or heterogeneous array of `poll_ctl` and `poll_ctl_ext` structures that describes the file descriptors (through the `pollfd` or `pollfd_ext` structure) and the unique operation to perform on each file descriptor (add, remove, or modify).
polldata_array	Returns the requested events that have occurred on the pollset.
pollfd_query	Points to a file descriptor that might or might not belong to the pollset. If it belongs to the pollset, then the requested *events* field of this parameter is updated to reflect what is currently being monitored for this file descriptor.
ps	Specifies the pollset ID.
timeout	Specifies the amount of time in milliseconds to wait for any monitored events to occur. A value of `-1` blocks until some monitored event occurs.

Return Values

Upon success, the `pollset_destroy` subroutine returns `0`. Upon failure, the `pollset_destroy` subroutine returns `-1` with `errno` set to the appropriate code.

Upon success, the `pollset_create` subroutine returns a pollset ID of type `pollset_t`. Upon failure, this system call returns `-1` with `errno` set appropriately.

Upon success, `pollset_ctl` or `pollset_ctl_ext` returns `0`. Upon failure, `pollset_ctl` or `pollset_ctl_exit` returns the 0-based problem element number of the `pollctl_array` (for example, 2 is returned for element 3). If the first element is the problem element, or some other error occurs prior to processing the array of elements, `-1` is returned and `errno` is set to the appropriate code. The calling application must acknowledge that elements in the array prior to the problem element were successfully processed and should attempt to call `pollset_ctl` or `pollset_ctl_ext` again with the elements of `pollctl_array` beyond the problematic element.

If a file descriptor is not a member of the pollset, `pollset_query` or `pollset_query_ext` returns `0`. If the file descriptor is in the pollset, `pollset_query` or `pollset_query_ext` returns 1, *events* is set to the currently monitored events, and when present, the *data* field is set to the user-specified data. If an error occurs after there is an attempt to determine if the file descriptor is a member of the pollset, then `pollset_query` or `pollset_query_ext` returns `-1` with `errno` set to the appropriate return code.

The `pollset_poll` or `pollset_poll_ext` subroutine returns the number of file descriptors on which requested events occurred. When no requested events occurred on any of the file descriptors, `0` is returned. A value of `-1` is returned when an error occurs and `errno` is set to the appropriate code.

Error Codes

Item	Description
EACCES	Process does not have permission to access a pollset.
EAGAIN	System resource temporarily not available.
EFAULT	Address supplied was not valid.
EINTR	A signal was received during the system call.
EINVAL	Invalid parameter.
ENOMEM	Insufficient system memory available.
ENOSPC	Maximum number of pollsets in use.
EPERM	Process does not have permission to create a pollset.
ENOTSUP	Device does not support the event combination requested.

popen Subroutine

Purpose

Initiates a pipe to a process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>

FILE *popen ( Command,  Type)
const char *Command, *Type;
```

Description

The **popen** subroutine creates a pipe between the calling program and a shell command to be executed.

Note: The **popen** subroutine runs only **sh** shell commands. The results are unpredictable if the *Command* parameter is not a valid **sh** shell command. If the terminal is in a trusted state, the **tsh** shell commands are run.

If streams opened by previous calls to the **popen** subroutine remain open in the parent process, the **popen** subroutine closes them in the child process.

The **popen** subroutine returns a pointer to a **FILE** structure for the stream.

Attention: If the original processes and the process started with the **popen** subroutine concurrently read or write a common file, neither should use buffered I/O. If they do, the results are unpredictable.

Some problems with an output filter can be prevented by flushing the buffer with the **fflush** subroutine.

Parameters

Item	Description
Command	Points to a null-terminated string containing a shell command line.

Item	Description
Type	Points to a null-terminated string containing an I/O mode. If the *Type* parameter is the value **r**, you can read from the standard output of the command by reading from the file *Stream*. If the *Type* parameter is the value **w**, you can write to the standard input of the command by writing to the file *Stream*. Because open files are shared, a type **r** command can be used as an input filter and a type **w** command as an output filter.

Return Values

The **popen** subroutine returns a null pointer if files or processes cannot be created, or if the shell cannot be accessed.

Error Codes

The **popen** subroutine may set the **EINVAL** variable if the *Type* parameter is not valid. The **popen** subroutine may also set **errno** global variables as described by the **fork** or **pipe** subroutines.

posix_fadvise Subroutine

Purpose

Provides advisory information to the system about the future behavior of the application with respect to a given file.

Syntax

```
 #include <fcntl.h>
int posix_fadvise (int fd, off_t offset, size_t len, int advice);
```

Description

This function advises the system on the expected future behavior of the application with regards to a given file. The system can take this advice into account when performing operations on file data specified by this function. The advice is given over the range covered by the *offset* parameter and continuing for the number of bytes specified by the *len* parameter. If the value of the *len* parameter is `0`, then all data following the *offset* parameter is covered.

To use the **posix_fadvise** subroutine, you must first open the file, and then call the **posix_fadvise** subroutine. The advisory information of a file is not reset when the file is closed. The client application must call the **posix_fadvise** subroutine along with the **POSIX_FADV_NORMAL** flag to reset all advisory information.

The *advice* parameter must have one of the following values:

POSIX_FADV_NORMAL
Resets all advisory information of a file to its default values.

POSIX_FADV_SEQUENTIAL
Valid option, but this value does not perform any action.

POSIX_FADV_RANDOM
Valid option, but this value does not perform any action.

POSIX_FADV_WILLNEED
Valid option, but this value does not perform any action.

POSIX_FADV_DONTNEED
Valid option, but this value does not perform any action.

POSIX_FADV_NOREUSE
Valid option, but this value does not perform any action.

POSIX_FADV_NOWRITEBEHIND
Instructs a file to ignore the normal write-behind functionality. You can run a system call, such as the sync system call, to explicitly write-back the information present in the file to the disk. This parameter value can be used only for regular files in enhanced Journaled File System (JFS2).

Parameters

Item	Description
fd	File descriptor of the file to be advised.
offset	Represents the beginning of the address range.
len	Determines the length of the address range.
advice	Defines the advice to be provided.

Return Values

Upon successful completion, the **posix_fadvise** subroutine returns 0. Otherwise, one of the following error codes will be returned.

Error Codes

Item	Description
EBADF	The *fd* parameter is not a valid file descriptor.
EINVAL	The value of the *advice* parameter is invalid.
ESPIPE	The *fd* parameter is associated with a pipe of FIFO.

posix_fallocate Subroutine

Purpose

Reserve storage space for a given file descriptor.

Syntax

```
 #include <fcntl.h>
int posix_fallocate (int fd, off_t offset, off_t len);
```

Description

This function reserves adequate space on the file system for the file data range beginning at the value specified by the *offset* parameter and continuing for the number of bytes specified by the *len* parameter. Upon successful return, subsequent writes to this file data range will not fail due to lack of free space on the file system media. Space allocated by the **posix_fallocate** subroutine can be freed by a successful call to the **creat** subroutine or **open** subroutine, or by the **ftruncate** subroutine, which truncates the file size to a size smaller than the sum of the *offset* parameter and the *len* parameter.

Note: Incase of return error code **EFBIG** and **ENOSPC**, the **posix_fallocate** subroutine might do partial allocation based on maximum file size or free space available on the file system.

Parameters

Item	Description
fd	File descriptor of the file toreserve
offset	Represents the beginning of the address range
len	Determines the length of the address range

Return Values

Upon successful completion, the **posix_fallocate** subroutine returns 0. Otherwise, one of the following error codes will be returned.

Error Codes

Item	Description
EBADF	The *fd* parameter is not a valid file descriptor
EBADF	The *fd* parameter references a file that was opened without write permission.
EFBIG	The value of the *offset* parameter plus the *len* parameter is greater than the maximum file size
EINTR	A signal was caught during execution
EIO	An I/O error occurred while reading from or writing to a file system
ENODEV	The *fd* parameter does not refer to a regular file.
EINVAL	The value of the *advice* parameter is invalid.
ENOSPC	There is insufficient free space remaining on the file system storage media
ESPIPE	The *fd* parameter is associated with a pipe of FIFO
ENOTSUP	The underlying file system is not supported

posix_madvise Subroutine

Purpose

Provides advisory information to the system regarding future behavior of the application with respect to a given range of memory.

Syntax

```
#include <sys/mman.h>
int posix_madvise (void *addr, size_t len, int advice);
```

Description

This function advises the system on the expected future behavior of the application with regard to a given range of memory. The system can take this advice into account when performing operations on the data in memory specified by this function. The advice is given over the range covered by the *offset* parameter and continuing for the number of bytes specified by the *addr* parameter and continuing for the number of bytes specified by the *len* parameter.

The *advice* parameter must be one of the following:

- **POSIX_MADV_NORMAL**
- **POSIX_MADV_SEQUENTIAL**
- **POSIX_MADV_RANDOM**
- **POSIX_MADV_WILLNEED**
- **POSIX_MADV_DONTNEED**

Parameters

Item	Description
addr	Defines the beginning of the range of memory to be advised
len	Determines the length of the address range
advice	Defines the advice to be given

Return Values

Upon successful completion, the **posix_fadvise** subroutine returns `0`. Otherwise, one of the following error codes will be returned.

Error Codes

Item	Description
EINVAL	The value of the *advice* parameter is invalid
ENOMEM	Addresses in the range specified by the *addr* parameter and the *len* parameter are partially or completely outside the range of the process's address space.

posix_openpt Subroutine

Purpose

Opens a pseudo-terminal device.

Library

Standard C library (**libc.a**)

Syntax

```
#include <stdlib.h<
#include <fcntl.h>

 int posix_openpt (oflag
)
int oflag;
```

Description

The **posix_openpt** subroutine establishes a connection between a controller device for a pseudo terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo terminal.

The file status flags and file access modes of the open file description are set according to the value of the *oflag* parameter.

Parameters

Item	Description
oflag	Values for the *oflag* parameter are constructed by a bitwise-inclusive OR of flags from the following list, defined in the **<fcntl.h>** file: **O_RDWR** Open for reading and writing. **O_NOCTTY** If set, the **posix_openpt** subroutine does not cause the terminal device to become the controlling terminal for the process. The behavior of other values for the *oflag* parameter is unspecified.

Return Values

Upon successful completion, the **posix_openpt** subroutine opens a controller pseudo-terminal device and returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **posix_openpt** subroutine will fail if:

Item	Description
EMFILE	OPEN_MAX file descriptors are currently open in the calling process.
ENFILE	The maximum allowable number of files is currently open in the system.

The **posix_openpt** subroutine may fail if:

Item	Description
EINVAL	The value of the *oflag* parameter is not valid.
EAGAIN	Out of pseudo-terminal resources.
ENOSR	Out of STREAMS resources.

Examples

The following example describes how to open a pseudo-terminal and return the name of the worker device and file descriptor

```
#include <fcntl.h>
#include <stdio.h>

int controllerfd, workerfd;
char *workerdevice;

controllerfd = posix_openpt(O_RDWR|O_NOCTTY);

if (controllerfd == -1
   || grantpt (controllerfd) == -1
      || unlockpt (controllerfd) == -1
      || (workerdevice = ptsname (controllerfd)) == NULL)
      return -1;

printf("worker device is: %s\n", workerdevice);

workerfd = open(workerdevice, O_RDWR|O_NOCTTY);
```

```
if (workerfd < 0)
   return -1;
```

posix_spawn or posix_spawnp Subroutine

Purpose

Spawns a process.

Syntax

```
int posix_spawn(pid_t *restrict pid, const char *restrict path,
       const posix_spawn_file_actions_t *file_actions,
       const posix_spawnattr_t *restrict attrp,
       char *const argv[restrict], char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
       const posix_spawn_file_actions_t *file_actions,
       const posix_spawnattr_t *restrict attrp,
       char *const argv[restrict], char * const envp[restrict]);
```

Description

The posix_spawn and posix_spawnp subroutines create a new process (child process) from the specified process image. The new process image is constructed from a regular executable file called the *new process image file*.

When a C program is executed as the result of this call, the program is entered as a C-language function call as follows:

```
int main(int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings.

The *argv* parameter is an array of character pointers to null-terminated strings. The last member of this array is a null pointer and is not counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a file name that is associated with the process image being started by the posix_spawn or posix_spawnp function.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The number of bytes available for the child process' combined argument and environment lists is {ARG_MAX}. The implementation specifies in the system documentation whether any list overhead, such as length words, null terminators, pointers, or alignment bytes, is included in this total.

The path argument to posix_spawn is a path name that identifies the new process image file to execute.

The file parameter to posix_spawnp is used to construct a path name that identifies the new process image file. If the file parameter contains a slash character (/), the file parameter is used as the path name for the new process image file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable PATH. If this environment variable is not defined, the results of the search are implementation-defined.

If *file_actions* is a null pointer, file descriptors that are open in the calling process remain open in the child process, except for those whose FD_CLOEXEC flag is set. For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks, remain unchanged.

If *file_actions* is not a null pointer, the file descriptors open in the child process are those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions is as follows:

1. The set of open file descriptors for the child process is initially the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks, remain unchanged.
2. The signal mask, signal default actions, and the effective user and group IDs for the child process are changed as specified in the attributes object referenced by *attrp*.
3. The file actions specified by the spawn file actions object are performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its FD_CLOEXEC flag set is closed.

The posix_spawnattr_t spawn attributes object type is defined in the spawn.h header file. Its attributes are defined as follows:

- If the POSIX_SPAWN_SETPGROUP flag is set in the spawn-flags attribute of the object referenced by *attrp*, and the spawn-pgroup attribute of the same object is non-zero, the child's process group is as specified in the spawn-pgroup attribute of the object referenced by *attrp*.
- As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the spawn-flags attribute of the object referenced by *attrp*, and the spawn-pgroup attribute of the same object is set to 0, then the child is in a new process group with a process group ID equal to its process ID.
- If the POSIX_SPAWN_SETPGROUP flag is not set in the spawn-flags attribute of the object referenced by *attrp*, the new child process inherits the parent's process group.
- If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the spawn-flags attribute of the object referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image initially has the scheduling policy of the calling process with the scheduling parameters specified in the spawn-schedparam attribute of the object referenced by *attrp*.
- If the POSIX_SPAWN_SETSCHEDULER flag is set in the spawn-flags attribute of the object referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image initially has the scheduling policy specified in the spawn-schedpolicy attribute of the object referenced by *attrp* and the scheduling parameters specified in the spawn-schedparam attribute of the same object.
- The POSIX_SPAWN_RESETIDS flag in the spawn-flags attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process inherits the parent process' effective user ID. If this flag is set, the child process' effective user ID is reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process becomes that file's owner ID before the new process image begins execution.
- The POSIX_SPAWN_RESETIDS flag in the spawn-flags attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process inherits the parent process' effective group ID. If this flag is set, the child process' effective group ID is reset to the parent's real group ID. In either case, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the child process becomes that file's group ID before the new process image begins execution.
- If the POSIX_SPAWN_SETSIGMASK flag is set in the spawn-flags attribute of the object referenced by *attrp*, the child process initially has the signal mask specified in the spawn-sigmask attribute of the object referenced by *attrp*.
- If the POSIX_SPAWN_SETSIGDEF flag is set in the spawn-flags attribute of the object referenced by *attrp*, the signals specified in the spawn-sigdefault attribute of the same object is set to their default actions in the child process. Signals set to the default action in the parent process are set to the default action in the child process. Signals set to be caught by the calling process are set to the default action in the child process.

- Except for SIGCHLD, signals set to be ignored by the calling process image are set to be ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the spawn-flags attribute of the object referenced by *attrp* and the signals being indicated in the spawn-sigdefault attribute of the object referenced by *attrp*.
- If the SIGCHLD signal is set to be ignored by the calling process, it is unspecified whether the SIGCHLD signal is set to be ignored or set to the default action in the child process. This is true unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the spawn_flags attribute of the object referenced by *attrp* and the SIGCHLD signal being indicated in the spawn_sigdefault attribute of the object referenced by *attrp*.
- If the value of the *attrp* pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by *attrp* in the preceding list or by the file descriptor manipulations specified in *file_actions*, are displayed in the new process image as though fork had been called to create a child process and then a member of the exec family of functions had been called by the child process to execute the new process image.

By default, fork handlers are not run in posix_spawn or posix_spawnp routines. To enable fork handlers, set the POSIX_SPAWN_FORK_HANDLERS flag in the attribute.

Return Values

Upon successful completion, posix_spawn and posix_spawnp return the process ID of the child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and return 0 as the function return value. Otherwise, no child process is created, the value stored into the variable pointed to by a non-NULL *pid* is unspecified, and an error number is returned as the function return value to indicate the error. If the *pid* argument is a null pointer, the process ID of the child is not returned to the caller.

Error Codes

The posix_spawn and posix_spawnp subroutines will fail if the following is true:

Item	Description
EINVAL	The value specified by *file_actions* or *attrp* is invalid.

The error codes for the posix_spawn and posix_spawnp subroutines are affected by the following conditions:

- If this error occurs after the calling process successfully returns from the posix_spawn or posix_spawnp function, the child process might exit with exit status 127.
- If posix_spawn or posix_spawnp fail for any of the reasons that would cause fork or one of the exec family of functions to fail, an error value is returned as described by fork and exec, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).
- If POSIX_SPAWN_SETPGROUP is set in the spawn-flags attribute of the object referenced by *attrp*, and posix_spawn or posix_spawnp fails while changing the child's process group, an error value is returned as described by setpgid (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).
- If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set in the spawn-flags attribute of the object referenced by *attrp*, then if posix_spawn or posix_spawnp fails for any of the reasons that would cause sched_setparam to fail, an error value is returned as described by sched_setparam (or, if the error occurs after the calling process successfully returns, the child process sexit with exit status 127).
- If POSIX_SPAWN_SETSCHEDULER is set in the spawn-flags attribute of the object referenced by *attrp*, and if posix_spawn or posix_spawnp fails for any of the reasons that would cause sched_setscheduler to fail, an error value is returned as described by sched_setscheduler (or,

if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

- If the *file_actions* argument is not NULL and specifies any close, dup2, or open actions to be performed, and if posix_spawn or posix_spawnp fails for any of the reasons that would cause close, dup2, or open to fail, an error value is returned as described by close, dup2, and open, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127). An open file action might, by itself, result in any of the errors described by close or dup2, in addition to those described by open.

posix_spawn_file_actions_addclose or posix_spawn_file_actions_addopen Subroutine

Purpose

Adds close or open action to the spawn file actions object.

Syntax

```
#include <spawn.h>

int posix_spawn_file_actions_addclose(posix_spawn_file_actions_t *
       file_actions, int fildes);
int posix_spawn_file_actions_addopen(posix_spawn_file_actions_t *
       restrict file_actions, int fildes,
       const char *restrict path, int oflag, mode_t mode);
```

Description

The posix_spawn_file_actions_addclose and posix_spawn_file_actions_addopen subroutines close or open action to a spawn file actions object.

A spawn file actions object is of type posix_spawn_file_actions_t (defined in the spawn.h header file) and is used to specify a series of actions to be performed by a posix_spawn or posix_spawnp operation in order to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent. Comparison or assignment operators for the type posix_spawn_file_actions_t are not defined.

A spawn file actions object, when passed to posix_spawn or posix_spawnp, specifies how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation is as if the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to running the new process image), in the order in which the actions were added to the object. Additionally, when the new process image is run, any file descriptor (from this new set) that has its FD_CLOEXEC flag set is closed (see “posix_spawn or posix_spawnp Subroutine” on page 1373).

The posix_spawn_file_actions_addclose function adds a close action to the object referenced by *file_actions* that causes the file descriptor *fildes* to be closed (as if close(*fildes*) had been called) when a new process is spawned using this file actions object.

The posix_spawn_file_actions_addopen function adds an open action to the object referenced by *file_actions* that causes the file named by path to be opened, as if open(*path*, *oflag*, *mode*) had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*) when a new process is spawned using this file actions object. If *fildes* was already an open file descriptor, it closes before the new file is opened.

The string described by *path* is copied by the posix_spawn_file_actions_addopen function.

Return Values

Upon successful completion, the `posix_spawn_file_actions_addclose` and `posix_spawn_file_actions_addopen` subroutines return `0`; otherwise, an error number is returned to indicate the error.

Error Codes

The `posix_spawn_file_actions_addclose` and `posix_spawn_file_actions_addopen` subroutines fail if the following is true:

Item	Description
EBADF	The value specified by *fildes* is negative, or greater than or equal to {OPEN_MAX}.

The `posix_spawn_file_actions_addclose` and `posix_spawn_file_actions_addopen` subroutines might fail if the following are true:

Item	Description
EINVAL	The value specified by *file_actions* is invalid.
ENOMEM	Insufficient memory exists to add to the spawn file actions object.

It is not an error for the *fildes* argument passed to these functions to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is used later during a `posix_spawn` or `posix_spawnp` operation.

posix_spawn_file_actions_adddup2 Subroutine

Purpose

Adds dup2 action to the spawn file actions object.

Syntax

```
#include <spawn.h>

int posix_spawn_file_actions_adddup2(posix_spawn_file_actions_t *
       file_actions, int fildes, int newfildes);
```

Description

The `posix_spawn_file_actions_adddup2` subroutine adds a dup2 action to the object referenced by *file_actions* that causes the file descriptor *fildes* to be duplicated as *newfildes* when a new process is spawned using this file actions object. This functions as if dup2(*fildes*, *newfildes*) had been called.

A spawn file actions object is as defined in `posix_spawn_file_actions_addclose`.

Return Values

Upon successful completion, the `posix_spawn_file_actions_adddup2` subroutine returns `0`; otherwise, an error number is returned to indicate the error.

Error Codes

The `posix_spawn_file_actions_adddup2` subroutine will fail if the following are true:

Item	Description
EBADF	The value specified by *fildes* or *newfildes* is negative, or greater than or equal to {OPEN_MAX}.
ENOMEM	Insufficient memory exists to add to the spawn file actions object.

The `posix_spawn_file_actions_adddup2` subroutine might fail if the following is true:

Item	Description
EINVAL	The value specified by *file_actions* is invalid.

It is not an error for the *fildes* argument passed to this subroutine to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is used later during a `posix_spawn` or `posix_spawnp` operation.

posix_spawn_file_actions_destroy or posix_spawn_file_actions_init Subroutine

Purpose

Destroys and initializes a spawn file actions object.

Syntax

```
#include <spawn.h>

int posix_spawn_file_actions_destroy(posix_spawn_file_actions_t *
       file_actions);
int posix_spawn_file_actions_init(posix_spawn_file_actions_t *
       file_actions);
```

Description

The `posix_spawn_file_actions_destroy` subroutine destroys the object referenced by *file_actions*; the object becomes, in effect, uninitialized. An implementation can cause `posix_spawn_file_actions_destroy` to set the object referenced by *file_actions* to an invalid value. A destroyed spawn file actions object can be reinitialized using `posix_spawn_file_actions_init`; the results of otherwise referencing the object after it has been destroyed are undefined.

The `posix_spawn_file_actions_init` function initializes the object referenced by *file_actions* to contain no file actions for `posix_spawn` or `posix_spawnp` to perform.

A spawn file actions object is as defined in `posix_spawn_file_actions_addclose`. The effect of initializing a previously initialized spawn file actions object is undefined.

Return Values

Upon successful completion, the `posix_spawn_file_actions_destroy` and `posix_spawn_file_actions_init` subroutines return `0`; otherwise, an error number is returned to indicate the error.

Error Codes

The `posix_spawn_file_actions_init` subroutine will fail if the following is true:

Item	Description
ENOMEM	Insufficient memory exists to initialize the spawn file actions object.

The posix_spawn_file_actions_destroy subroutine might fail if the following is true:

Item	Description
EINVAL	The value specified by *file_actions* is invalid.

posix_spawnattr_destroy or posix_spawnattr_init Subroutine

Purpose

Destroys and initializes a spawn attributes object.

Syntax

```
#include <spawn.h>

int posix_spawnattr_destroy(posix_spawnattr_t *attr);
int posix_spawnattr_init(posix_spawnattr_t *attr);
```

Description

The posix_spawnattr_destroy subroutine destroys a spawn attributes object. A destroyed *attr* attributes object can be reinitialized using posix_spawnattr_init; the results of otherwise referencing the object after it has been destroyed are undefined. An implementation can cause posix_spawnattr_destroy to set the object referenced by *attr* to an invalid value.

The posix_spawnattr_init subroutine initializes a spawn attributes object *attr* with the default value for all of the individual attributes used by the implementation. Results are undefined if posix_spawnattr_init is called specifying an *attr* attributes object that is already initialized.

A spawn attributes object is of type posix_spawnattr_t (defined in the spawn.h header file) and is used to specify the inheritance of process attributes across a spawn operation. Comparison or assignment operators for the type posix_spawnattr_t are not defined.

Each implementation documents the individual attributes it uses and their default values unless these values are defined by IEEE Std 1003.1-2001. Attributes not defined by IEEE Std 1003.1-2001, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

The resulting spawn attributes object (possibly modified by setting individual attribute values), is used to modify the behavior of posix_spawn or posix_spawnp. After a spawn attributes object has been used to spawn a process by a call to a posix_spawn or posix_spawnp, any function affecting the attributes object (including destruction) will not affect any process that has been spawned in this way.

Return Values

Upon successful completion, the posix_spawnattr_destroy and posix_spawnattr_init subroutines return 0; otherwise, an error number is returned to indicate the error.

Error Codes

The posix_spawnattr_destroy subroutine might fail if the following is true:

Item	Description
EINVAL	The value specified by *attr* is invalid.

posix_spawnattr_getflags or posix_spawnattr_setflags Subroutine

Purpose

Gets and sets the spawn-flags attribute of a spawn attributes object.

Syntax

```
#include <spawn.h>

int posix_spawnattr_getflags(const posix_spawnattr_t *restrict attr,
       short *restrict flags);
int posix_spawnattr_setflags(posix_spawnattr_t *attr, short flags);
```

Description

The posix_spawnattr_getflags subroutine obtains the value of the spawn-flags attribute from the attributes object referenced by *attr*. The posix_spawnattr_setflags subroutine sets the spawn-flags attribute in an initialized attributes object referenced by *attr*. The spawn-flags attribute is used to indicate which process attributes are to be changed in the new process image when invoking posix_spawn or posix_spawnp. It is the bitwise-inclusive OR of 0 or more of the following flags:

- POSIX_SPAWN_RESETIDS
- POSIX_SPAWN_SETPGROUP
- POSIX_SPAWN_SETSIGDEF
- POSIX_SPAWN_SETSIGMASK
- POSIX_SPAWN_SETSCHEDPARAM
- POSIX_SPAWN_SETSCHEDULER

These flags are defined in the spawn.h header file. The default value of this attribute is as if no flags were set.

Return Values

Upon successful completion, the posix_spawnattr_getflags subroutine returns 0 and stores the value of the spawn-flags attribute of *attr* into the object referenced by the *flags* parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the posix_spawnattr_setflags subroutine returns 0; otherwise, an error number is returned to indicate the error.

Error Codes

The posix_spawnattr_getflags and posix_spawnattr_setflags subroutines will fail if the following is true:

Item	Description
EINVAL	The value specified by *attr* is invalid.

The posix_spawnattr_setflags subroutine might fail if the following is true:

Item	Description
EINVAL	The value of the attribute being set is not valid.

posix_spawnattr_getpgroup or posix_spawnattr_setpgroup Subroutine

Purpose

Gets and sets the spawn-pgroup attribute of a spawn attributes object.

Syntax

```
#include <spawn.h>

int posix_spawnattr_getpgroup(const posix_spawnattr_t *restrict attr,
       pid_t *restrict pgroup);
int posix_spawnattr_setpgroup(posix_spawnattr_t *attr, pid_t pgroup);
```

Description

The posix_spawnattr_getpgroup subroutine gets the value of the spawn-pgroup attribute from the attributes object referenced by *attr*.

The posix_spawnattr_setpgroup subroutine sets the spawn-pgroup attribute in an initialized attributes object referenced by *attr*.

The spawn-pgroup attribute represents the process group to be joined by the new process image in a spawn operation (if POSIX_SPAWN_SETPGROUP is set in the spawn-flags attribute). The default value of this attribute is 0.

Return Values

Upon successful completion, the posix_spawnattr_getpgroup subroutine returns 0 and stores the value of the spawn-pgroup attribute of *attr* into the object referenced by the *pgroup* parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the posix_spawnattr_setpgroup subroutine returns 0; otherwise, an error number is returned to indicate the error.

Error Codes

The posix_spawnattr_getpgroup and posix_spawnattr_setpgroup subroutines might fail if the following is true:

Item	Description
EINVAL	The value specified by *attr* is invalid.

The posix_spawnattr_setpgroup subroutine might fail if the following is true:

Item	Description
EINVAL	The value of the attribute being set is not valid.

posix_spawnattr_getschedparam or posix_spawnattr_setschedparam Subroutine

Purpose

Gets and sets the spawn-schedparam attribute of a spawn attributes object.

Syntax

```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedparam(const posix_spawnattr_t *
       restrict attr, struct sched_param *restrict schedparam);
int posix_spawnattr_setschedparam(posix_spawnattr_t *restrict attr,
       const struct sched_param *restrict schedparam);
```

Description

The `posix_spawnattr_getschedparam` subroutine gets the value of the `spawn-schedparam` attribute from the attributes object referenced by *attr*.

The `posix_spawnattr_setschedparam` subroutine sets the `spawn-schedparam` attribute in an initialized attributes object referenced by *attr*.

The `spawn-schedparam` attribute represents the scheduling parameters to be assigned to the new process image in a spawn operation (if `POSIX_SPAWN_SETSCHEDULER` or `POSIX_SPAWN_SETSCHEDPARAM` is set in the `spawn-flags` attribute). The default value of this attribute is unspecified.

Return Values

Upon successful completion, the `posix_spawnattr_getschedparam` subroutine returns `0` and stores the value of the `spawn-schedparam` attribute of *attr* into the object referenced by the *schedparam* parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the `posix_spawnattr_setschedparam` subroutine returns `0`; otherwise, an error number is returned to indicate the error.

Error Codes

The `posix_spawnattr_getschedparam` and `posix_spawnattr_setschedparam` subroutines might fail if the following is true:

Item	Description
`EINVAL`	The value specified by *attr* is invalid.

The `posix_spawnattr_setschedparam` subroutine might fail if the following is true:

Item	Description
`EINVAL`	The value of the attribute being set is not valid.

posix_spawnattr_getschedpolicy or posix_spawnattr_setschedpolicy Subroutine

Purpose

Gets and sets the `spawn-schedpolicy` attribute of a spawn attributes object.

Syntax

```
#include <spawn.h>
#include <sched.h>

int posix_spawnattr_getschedpolicy(const posix_spawnattr_t *
       restrict attr, int *restrict schedpolicy);
```

```
int posix_spawnattr_setschedpolicy(posix_spawnattr_t *attr,
       int schedpolicy);
```

Description

The posix_spawnattr_getschedpolicy subroutine gets the value of the spawn-schedpolicy attribute from the attributes object referenced by *attr*.

The posix_spawnattr_setschedpolicy subroutine sets the spawn-schedpolicy attribute in an initialized attributes object referenced by *attr*.

The spawn-schedpolicy attribute represents the scheduling policy to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER is set in the spawn-flags attribute). The default value of this attribute is unspecified.

Return Values

Upon successful completion, the posix_spawnattr_getschedpolicy subroutine returns 0 and stores the value of the spawn-schedpolicy attribute of *attr* into the object referenced by the *schedpolicy* parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, posix_spawnattr_setschedpolicy returns 0; otherwise, an error number is returned to indicate the error.

Error Codes

The following posix_spawnattr_getschedpolicy and posix_spawnattr_setschedpolicy subroutines might fail if the following is true:

Item	Description
EINVAL	The value specified by *attr* is invalid.

The posix_spawnattr_setschedpolicy subroutine might fail if the following is true:

Item	Description
EINVAL	The value of the attribute being set is not valid.

posix_spawnattr_getsigdefault or posix_spawnattr_setsigdefault Subroutine

Purpose

Gets and sets the spawn-sigdefault attribute of a spawn attributes object.

Syntax

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigdefault(const posix_spawnattr_t *
       restrict attr, sigset_t *restrict sigdefault);
int posix_spawnattr_setsigdefault(posix_spawnattr_t *restrict attr,
       const sigset_t *restrict sigdefault);
```

Description

The posix_spawnattr_getsigdefault subroutine gets the value of the spawn-sigdefault attribute from the attributes object referenced by *attr*.

The posix_spawnattr_setsigdefault subroutine sets the spawn-pgroup attribute in an initialized attributes object referenced by *attr*.

The spawn-sigdefault attribute represents the set of signals to be forced to default signal handling in the new process image by a spawn operation (if POSIX_SPAWN_SETSIGDEF is set in the spawn-flags attribute). The default value of this attribute is an empty signal set.

Return Values

Upon successful completion, the posix_spawnattr_getsigdefault subroutine returns 0 and stores the value of the spawn-sigdefault attribute of *attr* into the object referenced by the *sigdefault* parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the posix_spawnattr_setsigdefault subroutine returns 0; otherwise, an error number is returned to indicate the error.

Error Codes

The posix_spawnattr_getsigdefault and posix_spawnattr_setsigdefault subroutines might fail if the following is true:

Item	Description
EINVAL	The value specified by *attr* is invalid.

The posix_spawnattr_setsigdefault subroutine might fail if the following is true:

Item	Description
EINVAL	The value of the attribute being set is not valid.

posix_spawnattr_getsigmask or posix_spawnattr_setsigmask Subroutine

Purpose

Gets and sets the spawn-sigmask attribute of a spawn attributes object.

Syntax

```
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_getsigmask(const posix_spawnattr_t *restrict attr,
       sigset_t *restrict sigmask);
int posix_spawnattr_setsigmask(posix_spawnattr_t *restrict attr,
       const sigset_t *restrict sigmask);
```

Description

The posix_spawnattr_getsigmask subroutine gets the value of the spawn-sigmask attribute from the attributes object referenced by *attr*.

The posix_spawnattr_setsigmask subroutine sets the spawn-sigmask attribute in an initialized attributes object referenced by *attr*.

The spawn-sigmask attribute represents the signal mask in effect in the new process image of a spawn operation (if POSIX_SPAWN_SETSIGMASK is set in the spawn-flags attribute). The default value of this attribute is unspecified.

Return Values

Upon successful completion, the `posix_spawnattr_getsigmask` subroutine returns `0` and stores the value of the `spawn-sigmask` attribute of *attr* into the object referenced by the *sigmask* parameter; otherwise, an error number is returned to indicate the error.

Upon successful completion, the `posix_spawnattr_setsigmask` subroutine returns `0`; otherwise, an error number is returned to indicate the error.

Error Codes

The `posix_spawnattr_getsigmask` and `posix_spawnattr_setsigmask` subroutines might fail if the following is true:

Item	Description
`EINVAL`	The value specified by *attr* is invalid.

The `posix_spawnattr_setsigmask` subroutine might fail if the following is true:

Item	Description
`EINVAL`	The value of the attribute being set is not valid.

posix_trace_attr_destroy Subroutine

Purpose

Destroys a trace stream attribute object.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_attr_destroy(attr)
trace_attr_t * attr;
```

Description

The `posix_trace_attr_destroy` subroutine destroys an initialized trace attributes object. A destroyed *attr* attributes object can be initialized again using the **posix_trace_attr_init** subroutine. The results of referencing the object after it has been destroyed are not defined.

If the `posix_trace_attr_destroy` subroutine is called with a non-initialized attributes object as a parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object to destroy.

Return Values

Upon successful completion, it returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The following error code return when the `posix_trace_attr_destroy` subroutine fails:

Item	Description
EINVAL	The value of the *attr* parameter is null.

Files

The `trace.h` file in *Files Reference*

posix_trace_attr_getcreatetime Subroutine

Purpose

Retrieves the creation time of a trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <time.h>
#include <trace.h>

int posix_trace_attr_getcreatetime(attr, createtime)
const trace_attr_t *attr;
struct timespec *createtime;
```

Description

The **posix_trace_attr_getcreatetime** subroutine copies the amount of time to create a trace stream from the *creation-time* attribute of the *attr* object into the *createtime* parameter. The value of the *createtime* parameter is a structure.

The `timespec` struct defines that the value of the *creation-time* attribute is a structure. The *creation-time* attribute is set with the `clock_gettime` subroutine. The `clock_gettime` subroutine returns the amount of time (in seconds and nanoseconds) since the epoch. The `timespec` struct is defined as the following:

```
struct timespec {
    time_t tv_sec;         /* seconds */
    long   tv_nsec;        /* and nanoseconds */
};
```

If the `posix_trace_attr_getcreatetime` subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
createtime	Specifies where the *creation-time* attribute is stored.

Return Values

Upon successful completion, it returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_attr_getcreatetime` subroutine stores the trace stream creation time in the *createtime* parameter. Otherwise, the content of this object is not specified.

Errors

The `posix_trace_attr_getcreatetime` subroutine fails if the following error number returns:

Item	Description
`EINVAL`	One of the parameters is null. Or the trace attributes object is not retrieved with the `posix_trace_get_attr` subroutine on a stream.

Files

The **trace.h** file in *Files Reference*

posix_trace_attr_getclockres Subroutine

Purpose

Retrieves the clock resolution.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <time.h>
#include <trace.h>

int posix_trace_attr_getclockres(attr, resolution)
const trace_attr_t *attr;
struct timespec *resolution;
```

Description

The **posix_trace_attr_getclockres** subroutine copies the clock resolution of the clock that is used to generate timestamps from the *attr* object into the *resolution* parameter. The *attr* object defines the clock resolution. The *resolution* parameter points to the structure.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
resolution	Specifies where the *clock-resolution* attribute of the *attr* object is stored.

Return Values

Upon successful completion, it returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_attr_getclockres` subroutine stores the clock-resolution attribute value of the *resolution* parameter. Otherwise, the content of this object is not specified.

Errors

The `posix_trace_attr_getclockres` subroutine fails if the following error number returns:

Item	Description
`EINVAL`	One of the parameters is null.

Files

The **trace.h** file in *Files Reference*

posix_trace_attr_getgenversion Subroutine

Purpose

Retrieves the version of a trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_attr_getgenversion(attr, genversion)
const trace_attr_t *attr;
char *genversion;
```

Description

The **posix_trace_attr_getgenversion** subroutine copies the string containing version information from the *version* attribute of the *attr* object into the *genversion* parameter. The *attr* parameter represents the generation version. The value of the *genversion* parameter points to a string. The *genversion* parameter is the address of a character array that can store at least the number of characters defined by the `TRACE_NAME_MAX` characters (see `limits.h` File).

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
genversion	Specifies where the *version* attribute is stored.

Return Values

Upon successful completion, it returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_attr_getgenversion` subroutine stores the trace version information in the string pointed to by the *genversion* parameter. Otherwise, the content of this string is not specified.

Errors

The `posix_trace_attr_getgenversion` subroutine fails if the following error number returns:

Item	Description
`EINVAL`	One of the parameters is null.

Files

The **trace.h** and the `limits.h` files in *Files Reference*

posix_trace_attr_getinherited Subroutine

Purpose

Retrieves the inheritance policy of a trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>
int posix_trace_attr_getinherited(attr,inheritancepolicy)
const trace_attr_t * attr;
int *restrict inheritancepolicy;
```

Description

The **posix_trace_attr_getinherited** subroutine gets the inheritance policy stored in the *inheritance* attribute of the *attr* object for traced processes across the `fork` and `posix_spawn` subroutine. The *inheritance* attribute of the *attr* object is set to one of the following values defined by manifest constants in the `trace.h` header file:

Item	Description
POSIX_TRACE_CLOSE_FOR_CHILD	After a **fork** or **spawn** operation, the child is not traced, and tracing of the parent continues.
POSIX_TRACE_INHERITED	After a **fork** or **spawn** operation, if the parent is being traced, its child will be simultaneously traced using the same trace stream.

The default value for of the *inheritance* attribute is *POSIX_TRACE_CLOSE_FOR_CHILD*.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attribute object.
inheritancepolicy	Specifies where the *inheritance* attribute of the *attr* object is stored.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_attr_getinherited` subroutine stores the value of the *attr* object in the object specified by the *inheritancepolicy* parameter. Otherwise, the content of this object is not modified.

Errors

This subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The object of a parameter is null or not valid.

Files

The `trace.h` file in the *Files Reference*

posix_trace_attr_getlogfullpolicy Subroutine

Purpose

Retrieves the log full policy of a trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>
int posix_trace_attr_getlogfullpolicy(attr,logpolicy)
const trace_attr_t *restrict;
int *restrict logpolicy;
```

Description

The `posix_trace_attr_getlogfullpolicy` subroutine gets the trace log full policy stored in the *log-full-policy* attribute of the *attr* object. The *attr* object points to the attribute object to get log full policy.

The *log-full-policy* attribute of the *attr* object is set to one of the following values defined by manifest constants in the `trace.h` header file:

Item	Description
POSIX_TRACE_LOOP	The trace log loops until the associated trace stream is stopped. When the trace log gets full, the file system reuses the resources allocated to the oldest trace events that were recorded. In this way, the trace log always contains the most recent trace events that are flushed.
POSIX_TRACE_UNTIL_FULL	The trace stream is flushed to the trace log until the trace log is full. This condition can be deduced from the *posix_log_full_status* member status (see the *posix_trace_status_info* structure defined in the `trace.h` header file). The last recorded trace event is the *POSIX_TRACE_STOP* trace event.

Item	Description
POSIX_TRACE_APPEND	The associated trace stream is flushed to the trace log without log size limitation. If the application specifies the *POSIX_TRACE_APPEND* value, the *log-max-size* attribute is ignored.

The default value for the *log-full-policy* attribute is *POSIX_TRACE_LOOP*.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attribute object.
logpolicy	Specifies where the *log-full-policy* attribute of the *attr* parameter is attained or stored.

Return Values

Upon successful completion, it returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_attr_getlogfullpolicy` subroutine stores the value of the *log-full-policy* attribute in the object specified by the *logpolicy* parameter. Otherwise, the content of this object is not modified.

Errors

The `posix_trace_attr_getlogfullpolicy` subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The object of a parameter is null or not valid.

Files

The trace.h file in *Files Reference*

posix_trace_attr_getlogsize Subroutine

Purpose

Retrieves the size of the log of a trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getlogsize(attr, logsize)
const trace_attr_t *restrict attr;
size_t *restrict logsize;
```

Description

The `posix_trace_attr_getlogsize` subroutine copies the size of a log in bytes from the *log-max-size* attribute of the *attr* parameter into the *logsize* variable. This size is the maximum total bytes that is allocated for system and user trace events in the trace log. The default value for the *attr* parameter is 1 MB.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attribute object.
logsize	Specifies where the *attr* parameter, in bytes, will be stored.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

The `posix_trace_attr_getlogsize` subroutine stores the maximum trace log size that is allowed in the object pointed to by the *logsize* parameter, if successful.

Errors

This subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The parameter is null or not valid.

Files

The `trace.h` file and the `types.h` file in *Files Reference*

posix_trace_attr_getmaxdatasize Subroutine

Purpose

Retrieves the maximum user trace event data size.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getmaxdatasize(attr, maxdatasize)
const trace_attr_t *restrict attr;
size_t *restrict maxdatasize;
```

Description

The `posix_trace_attr_getmaxdatasize` subroutine copies the maximum user trace event data size, in bytes, from the *max-data-size* attribute of the *attr* object into the variable specified the *maxdatasize* parameter. The default value for the *max-data-size* attribute is 16 bytes.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes' object.
maxdatasize	Specifies where the *max-data-size* attribute, in bytes, will be stored.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

The `posix_trace_attr_getmaxdatasize` subroutine stores the maximum trace event record memory size in the object pointed to by the *maxdatasize* parameter, if successful.

Errors

This subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The parameter is null or not valid.

Files

The `trace.h` file and the `types.h` file in *Files Reference*.

posix_trace_attr_getmaxsystemeventsize Subroutine

Purpose

Retrieves the maximum size of a system trace event.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getmaxsystemeventsize(attr, eventsize)
const trace_attr_t *restrict attr;
size_t *restrict eventsize;
```

Description

The `posix_trace_attr_getmaxsystemeventsize` subroutine calculates the maximum size, in bytes, of memory that is required to store a single system trace event. The size value is calculated for the trace stream attributes of the *attr* object, and is returned in the *eventsize* parameter.

The values returned as the maximum memory sizes of the user and system trace events, so that when the sum of the maximum memory sizes of a set of the trace events, which might be recorded in a trace stream, is less than or equal to the minimum stream size attribute of that trace stream, the system provides the necessary resources for recording all those trace events without loss.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
eventsize	Specifies where the maximum memory size attribute of the *attr* object, in bytes, will be stored.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

The `posix_trace_attr_getmaxsystemeventsize` subroutine stores the maximum memory size to store a single system trace event in the object pointed to by the *eventsize* parameter, if successful.

Errors

This subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The *attr* parameter is null or the other parameter is not valid.

Files

The `trace.h` file and the `types.h` file in the *Files Reference*

posix_trace_attr_getmaxusereventsize Subroutine

Purpose

Retrieves the maximum size of an user event for a given length.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getmaxusereventsize(attr, data_len, eventsize)
const trace_attr_t *restrict attr;
size_t data_len;
size_t *restrict eventsize;
```

Description

The `posix_trace_attr_getmaxusereventsize` subroutine calculates the maximum size, in bytes, of memory that is required to store a single user trace event that is generated by the `posix_trace_event`

subroutine with a *data_len* parameter equal to the *data_len* value specified in this subroutine. The size value is calculated for the trace stream attributes object pointed to by the *attr* parameter, and is returned in the variable specified by the *eventsize* parameter.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
data_len	Specifies the *data_len* parameter that is used to compute the maximum memory size that is required to stored a single user trace event.
eventsize	Specifies where the *attr* object, in bytes, will be stored.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

The `posix_trace_attr_getmaxusereventsize` subroutine stores the maximum memory size to store a single user trace event in the object pointed to by the *eventsize* parameter, if successful.

Errors

This subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The *attr* parameter is null or the other parameters are not valid.

Files

The `trace.h` file and the `types.h` file in the *Files Reference*

posix_trace_attr_getname Subroutine

Purpose

Retrieves the trace name.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_attr_getname(attr, tracename)
const trace_attr_t *attr;
char *tracename;
```

Description

The `posix_trace_attr_getname` subroutine copies the string containing the trace name from the *trace-name* attribute of the *attr* object into the *tracename* parameter. The *tracename* parameter points to

a string, and it is the address of a character array that can store at least TRACE_NAME_MAX characters (see limits.h File).

If the posix_trace_attr_getname subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
tracename	Specifies where the *trace-name* attribute is stored.

Return Values

Upon successful completion, the posix_trace_attr_getname subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the posix_trace_attr_getname subroutine stores the trace name in the string pointed to by the *tracename* parameter. Otherwise, the content of this string is not specified.

Errors

The posix_trace_attr_getname subroutine fails if the following error number returns:

Item	Description
EINVAL	One of the parameters is null.

Files

The **trace.h** and the limits.h Files in *Files Reference*

posix_trace_attr_getstreamfullpolicy Subroutine

Purpose

Retrieves the stream full policy.

Library

Posix Trace Library (libposixtrace.a)

Syntax

```
#include <trace.h>
int posix_trace_attr_getstreamfullpolicy(attr,streampolicy)
const trace_attr_t *attr;
int *streampolicy;
```

Description

The posix_trace_attr_getstreamfullpolicy subroutine gets the trace stream full policy stored in *stream-full-policy* attribute of the *attr* object.

The *stream-full-policy* attribute of the *attr* object is set to one of the following values defined by manifest constants in the trace.h header file:

Item	Description
POSIX_TRACE_LOOP	The trace stream loops until explicitly stopped by the `posix_trace_stop` subroutine. When the trace stream is full, the trace system reuses the resources allocated to the oldest trace events recorded. In this way, the trace stream always contains the most recent trace events that are recorded.
POSIX_TRACE_UNTIL_FULL	The trace stream runs until the trace stream resources are exhausted. This condition can be deduced from the *posix_stream_status* and *posix_stream_full_status* (see the *posix_trace_status_info* structure defined in `trace.h` header file). When this trace stream is read, a *POSIX_TRACE_STOP* trace event is reported after the last recorded trace event. The trace system reuses the resources that are allocated to any reported trace events (see the `posix_trace_getnext_event`, `posix_trace_trygetnext_event`, and `posix_trace_timedgetnext_event` subroutines), or trace events that are flushed for an active trace stream with log. The trace system restarts the trace stream when 50 per cent of the buffer size is read. A *POSIX_TRACE_START* trace event is reported before reporting the next recorded trace event.
POSIX_TRACE_FLUSH	This policy is identical to the *POSIX_TRACE_UNTIL_FULL* trace stream full policy except that the trace stream is flushed regularly as if the `posix_trace_flush` subroutine is called. Defining this policy for an active trace stream without log is not valid.

For an active trace stream without log, the default value for the *stream-full-policy* attribute is *POSIX_TRACE_LOOP*.

For an active trace stream with log, the default value for the*stream-full-policy* attribute is *POSIX_TRACE_FLUSH*.

If the subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
streampolicy	Specifies where the *stream-full-policy* attribute of the *attr* object is stored.

Return Values

Upon successful completion, the subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_attr_getstreamfullpolicy` subroutine stores the value of the *stream-full-policy* attribute in the object specified by the *streampolicy* parameter. Otherwise, the content of this object is not modified.

Errors

The subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The *attr* parameter is null or the other parameter is not valid.

Files

The `trace.h` file in *Files Reference*

posix_trace_attr_getstreamsize Subroutine

Purpose

Retrieves the trace stream size.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_getstreamsize(attr, streamsize)
trace_attr_t *attr;
size_t streamsize;
```

Description

The `posix_trace_attr_getstreamsize` subroutine copies the stream size, in bytes, from the *stream_minsize* attribute of the *attr* object into the variable pointed to by the *streamsize* parameter.

This stream size is the current total memory size reserved for system and user trace events in the trace stream. The default value for the *stream_minsize* attribute is 8192 bytes. The stream size refers to memory that is used to store trace event records. Other stream data (for example, trace attribute values) are not included in this size.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
streamsize	Specifies where the *stream_minsize* attribute, in bytes, will be stored.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

The `posix_trace_attr_getstreamsize` subroutine stores the maximum trace stream allowed size in the object pointed to by the *streamsize* parameter, if successful.

Errors

This subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The *attr* parameter is null or the other parameter is not valid.

Files

The `trace.h` file and the `types.h` file in the *Files Reference*

posix_trace_attr_init Subroutine

Purpose

Initializes a trace stream attributes object.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_attr_init(attr)
trace_attr_t * attr;
```

Description

The `posix_trace_attr_init` subroutine initializes a trace attributes object, the *attr* object, with the following default values :

Attribute field	Default value
stream_minsize	8192 bytes
stream_fullpolicy	For a stream without LOG, the default value is POSIX_TRACE_LOOP
	For a stream with LOG, the default value is POSIX_TRACE_FLUSH
max_datasize	16 bytes
inheritance	POSIX_TRACE_CLOSE_FOR_CHILD
log_maxsize	1 MB
log_fullpolicy	POSIX_TRACE_LOOP

The *version* and *clock-resolution* attributes that are generated by the initialized trace attributes object are set to the following values:

Attribute field	Value
version	0.1
clock-resolution	Clock resolution of the clock used to generate timestamps.

When the stream is created by the `posix_trace_create` or `posix_trace_create_withlog` subroutines, the *creation_time* attribute is set.

When the `posix_trace_attr_init` subroutine is called specifying an already initialized *attr* attributes object, this object is initialized with default values, the same as the values in the first initialization. If it is not saved, the already initialized *attr* attributes object is not accessible any more.

When used by the `posix_trace_create` subroutine, the resulting attributes object defines the attributes of the trace stream created. A single attributes object can be used in multiple calls to the `posix_trace_create` subroutine. After one or more trace streams have been created using an attributes object, any subroutine affecting that attributes object, including destruction, will not affect any

trace stream previously created. An initialized attributes object also serves to receive the attributes of an existing trace stream or trace log when calling the `posix_trace_get_attr` subroutine.

The **posix_trace_attr_init** subroutine initializes again a destroyed *attr* attributes object.

Parameters

Item	Description
attr	Specifies the trace attributes object to initialize.

Return Values

Upon successful completion, it returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The following error codes return when the `posix_trace_attr_init` subroutine fails:

Item	Description
`EINVAL`	The value of the *attr* parameter is null.
`ENOMEM`	Insufficient memory to initialize the trace attribute object .

Files

The `trace.h` file in *Files Reference*

posix_trace_attr_setinherited Subroutines

Purpose

Sets the inheritance policy of a trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>
int posix_trace_attr_setinherited(attr,inheritancepolicy)
const trace_attr_t * attr;
int *restrict inheritancepolicy;
```

Description

The **posix_trace_attr_setinherited** subroutine sets the inheritance policy stored in the *inheritance* attribute of the *attr* object for traced processes across the `fork` and `posix_spawn` subroutine. The *inheritance* attribute of the *attr* object is set to one of the following values defined by manifest constants in the `trace.h` header file:

Item	Description
POSIX_TRACE_CLOSE_FOR_CHILD	After a **fork** or **spawn** operation, the child is not traced, and tracing of the parent continues.

Item	Description
POSIX_TRACE_INHERITED	After a **fork** or **spawn** operation, if the parent is being traced, its child will be simultaneously traced using the same trace stream.

The default value for the *attr* object is *POSIX_TRACE_CLOSE_FOR_CHILD*.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies trace attributes object.
inheritancepolicy	Specifies where the *inheritance* attribute is attained.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following error number returns:

Item	Description
EINVAL	The *attr* parameter is null or the other parameter is not valid.

Files

The `trace.h` file in the *Files Reference*.

posix_trace_attr_setlogsize Subroutine

Purpose

Sets the size of the log of a trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setlogsize(attr, logsize)
const trace_attr_t *restrict attr;
size_t *restrict logsize;
```

Description

The `posix_trace_attr_setlogsize` subroutine sets the maximum allowed size in bytes in the *log-max-size* attribute of the *attr* object, using the size value specified by the *logsize* parameter. If the *logsize* parameter is too small regarding the stream size, the `posix_trace_attr_setlogsize` subroutine

does not fail. It sets the *log-max-size* attribute in order to be able to write at least one stream in the log file. Further calls to the `posix_trace_create` or `posix_trace_create_withlog` subroutines with such an attributes object will not fail.

The size of the trace log is used if the *log-full-policy* attribute of the *attr* object is set to the *POSIX_TRACE_LOOP* value or the *POSIX_TRACE_UNTIL_FULL* value. If the *attr* object is set to the *POSIX_TRACE_APPEND* value. The system ignores the *log-max-size* attribute in this case.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
logsize	Specifies where the *log-max-size* attribute, in bytes, will be attained.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The *attr* parameter is null or the other parameter is not valid.

Files

The `trace.h` file and the `types.h` file in *Files Reference*

posix_trace_attr_setmaxdatasize Subroutine

Purpose

Sets the maximum user trace event data size.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setmaxdatasize(attr, maxdatasize)
trace_attr_t *attr;
size_t maxdatasize;
```

Description

The `posix_trace_attr_setmaxdatasize` subroutine sets the maximum size, in bytes, that is allowed, in the *max-data-size* attribute of the *attr* object, using the size value specified by the *maxdatasize* parameter. This maximum size is the maximum allowed size for the user data argument

that could be passed to the posix_trace_event subroutine. The system truncates data passed to posix_trace_event the which is longer than the maximum data size.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
maxdatasize	Specifies where the *max-data-size* attribute, in bytes, will be attained.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following error number returns:

Item	Description
EINVAL	The *attr* parameter is null or the other parameter is not valid.

Files

The trace.h file and the types.h file in the *Files Reference*.

posix_trace_attr_setname Subroutine

Purpose

Sets the trace name.

Library

Posix Trace Library (libposixtrace.a)

Syntax

```
#include <trace.h>

int posix_trace_attr_setname(attr, tracename)
trace_attr_t *attr;
const char *tracename;
```

Description

The posix_trace_attr_setname subroutine sets the name in the *trace-name* attribute of the *attr* object with the string pointed to by the *tracename* parameter. If the length of the string name exceeds the value of the TRACE_NAME_MAX characters, the name copied into the *attr* object will be truncated to one that is less than the length of the TRACE_NAME_MAX characters (see limits.h File). The default value is a null string.

If the posix_trace_attr_setname subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
tracename	Specifies where the *trace-name* attribute is attained.

Return Values

Upon successful completion, the `posix_trace_attr_setname` subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The `posix_trace_attr_setname` subroutine fails if the following error number returns:

Item	Description
`EINVAL`	One of the parameters is null.

Files

The **trace.h** and the `limits.h` files in *Files Reference*

posix_trace_attr_setlogfullpolicy Subroutine

Purpose

Sets the log full policy of a trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>
int posix_trace_attr_setlogfullpolicy(attr,logpolicy)
const trace_attr_t *restrict;
int *restrict logpolicy;
```

Description

The `posix_trace_attr_setlogfullpolicy` subroutine sets the trace log full policy stored in *log-full-policy* attribute of the *attr* object. The *attr* parameter points to the attribute object to get log full policy.

The *log-full-policy* attribute of the *attr* parameter is set to one of the following values defined by manifest constants in the `trace.h` header file:

Item	Description
POSIX_TRACE_LOOP	The trace log loops until the associated trace stream is stopped. When the trace log gets full, the file system reuses the resources allocated to the oldest trace events that were recorded. In this way, the trace log always contains the most recent trace events that are flushed.

Item	Description
POSIX_TRACE_UNTIL_FULL	The trace stream is flushed to the trace log until the trace log is full. This condition can be deduced from the *posix_log_full_status* member status (see the *posix_trace_status_info* structure defined in the `trace.h` header file). The last recorded trace event is the *POSIX_TRACE_STOP* trace event.
POSIX_TRACE_APPEND	The associated trace stream is flushed to the trace log without log size limitation. If the application specifies the *POSIX_TRACE_APPEND* value, the *log-max-size* attribute is ignored.

The default value for the *log-full-policy* attribute is *POSIX_TRACE_LOOP*.

If the subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
logpolicy	Specifies where the *log-full-policy* attribute of the *attr* parameter is attained.

Return Values

Upon successful completion, the subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The *attr* parameter is null or the other parameter is not valid.

Files

The **trace.h** file in *Files Reference*

posix_trace_attr_setstreamfullpolicy Subroutine

Purpose

Sets the stream full policy.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>
int posix_trace_attr_setstreamfullpolicy(attr,streampolicy)
const trace_attr_t *attr;
int *streampolicy;
```

Description

The `posix_trace_attr_setstreamfullpolicy` subroutine sets the trace stream full policy stored in *stream-full-policy* attribute of the *attr* object.

The *stream-full-policy* attribute of the *attr* object is set to one of the following values defined by manifest constants in the `trace.h` header file:

Item	Description
POSIX_TRACE_LOOP	The trace stream loops until explicitly stopped by the `posix_trace_stop` subroutine. When the trace stream is full, the trace system reuses the resources allocated to the oldest trace events recorded. In this way, the trace stream always contains the most recent trace events that are recorded.
POSIX_TRACE_UNTIL_FULL	The trace stream runs until the trace stream resources are exhausted. This condition can be deduced from the *posix_stream_status* and *posix_stream_full_status* (see the *posix_trace_status_info* structure defined in `trace.h` header file). When this trace stream is read, a *POSIX_TRACE_STOP* trace event is reported after the last recorded trace event. The trace system reuses the resources that are allocated to any reported trace events (see the `posix_trace_getnext_event`, `posix_trace_trygetnext_event`, and `posix_trace_timedgetnext_event` subroutines), or trace events that are flushed for an active trace stream with log (see the `posix_trace_flush` subroutine). The trace system restarts the trace stream when 50 per cent of the buffer size is read. A *POSIX_TRACE_START* trace event is reported before reporting the next recorded trace event.
POSIX_TRACE_FLUSH	This policy is identical to the *POSIX_TRACE_UNTIL_FULL* trace stream full policy except that the trace stream is flushed regularly as if the `posix_trace_flush` subroutine is called. Defining this policy for an active trace stream without log is not valid.

For an active trace stream without log, the default value of the *stream-full-policy* attribute for the *attr* object is *POSIX_TRACE_LOOP*.

For an active trace stream with log, the default value of the *stream-full-policy* attribute for the *attr* object is *POSIX_TRACE_FLUSH*.

If the subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
streampolicy	Specifies where the *stream-full-policy* attribute of the *attr* object is attained.

Return Values

Upon successful completion, the subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The subroutine fails if the following error number returns:

Item	Description
EINVAL	The *attr* parameter is null or the other parameter is not valid.

Files

The `trace.h` file in *Files Reference*

posix_trace_attr_setstreamsize Subroutine

Purpose

Sets the trace stream size.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_attr_setstreamsize(attr, streamsize)
trace_attr_t *attr;
size_t streamsize;
```

Description

The `posix_trace_attr_setstreamsize` subroutine sets the minimum size that is allowed, in bytes, in the *stream_minsize* attribute of the *attr* object, using the size value specified by the *streamsize* parameter. If the *streamsize* parameter is smaller than the minimum required size, the `posix_trace_attr_setstreamsize` subroutine does not fail. It sets this minimum size in the *stream_minsize* attribute. Further calls to the `posix_trace_create`subroutine or the `posix_trace_create_withlog` subroutines will not fail.

If this subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
attr	Specifies the trace attributes object.
streamsize	Specifies where the *stream_minsize* attribute of the *attr* object, in bytes, will be attained.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The `posix_trace_attr_setstreamsize` subroutine fails if the following error number returns:

Item	Description
EINVAL	The requested size for the stream is larger than the segment size. The parameter is null or the other parameter is not valid.

Files

The trace.h file and the types.h file in the *Files Reference*

posix_trace_clear Subroutine

Purpose

Clears trace stream and trace log.

Library

Posix Trace Library (libposixtrace.a)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_clear(trid)
trace_id_t trid;
```

Description

The posix_trace_clear subroutine initializes the trace stream identified by the *trid* parameter again. It returns the same result as that of the posix_trace_create subroutine. The posix_trace_clear subroutine reuses the allocated resources of the posix_trace_create subroutine, but does not change the mapping of trace event type identifiers, which is used to trace event names, and it does not change the trace stream status.

All trace events in the trace stream recorded before the call to the posix_trace_clear subroutine are lost. The status of the posix_stream_full_status is set to the POSIX_TRACE_NOT_FULL status. There is no guarantee that all trace events that occurred during the posix_trace_clear call are recorded.

If the trace stream is created with a log, the posix_trace_clear subroutine initializes the trace stream with the same behavior again as if the trace stream was created without the log. It initializes the trace log associated with the trace stream identified by the *trid* parameter again. It uses the same allocated resources for the trace log of the posix_trace_create_withlog subroutine and the associated trace stream status remains unchanged. The first trace event recorded in the trace log after the call to the posix_trace_clear subroutine is the same as the first trace event recorded in the active trace stream after the call to posix_trace_clear subroutine. The posix_log_full_status status is set to POSIX_TRACE_NOT_FULL and the posix_log_overrun_status is set to POSIX_TRACE_NO_OVERRUN. There is no guarantee that all trace events that occurred during the posix_trace_clear call are recorded in the trace log. If the log full policy is POSIX_TRACE_APPEND, the stream and the trace log are initialized again as if it is returning from the posix_trace_withlog subroutine.

Parameters

Item	Description
trid	Specifies the trace stream identifier of an active trace stream.

Return Values

Upon successful completion, the posix_trace_clear subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

Item	Description
EINVAL	The value of the *trid* parameter does not correspond to an active trace stream.

Files

The **trace.h** and the **types.h** files in the *Files Reference*

posix_trace_close Subroutine

Purpose

Closes a trace log.

Library

Posix Trace Library (libposixtrace.a)

Syntax

```
#include <trace.h>

int posix_trace_close (trid)
trace_id_t trid;
```

Description

The posix_trace_close subroutine deallocates the trace log identifier indicated by the *trid* parameter, and all of its associated resources. If there is no valid trace log pointed to by the *trid* parameter, this subroutine fails.

Parameters

Item	Description
trid	Specifies the trace stream identifier.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The posix_trace_close subroutine fails if the following error returns:

Item	Description
EINVAL	The object pointed to by the *trid* parameter does not correspond to a valid trace log.

Files

The **trace.h** file in the *Files Reference*

posix_trace_create Subroutine

Purpose

Creates an active trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_create (pid, attr, trid)
pid_t pid;
const trace_attr_t *restrict attr;
trace_id_t *restrict trid;
```

Description

The `posix_trace_create` subroutine creates an active trace stream. It allocates all of the resources needed by the trace stream being created for tracing the process specified by the *pid* parameter in accordance with the *attr* parameter.

The *attr* parameter represents the initial attributes of the trace stream and must be initialized by the `posix_trace_attr_init` subroutine before the `posix_trace_create` subroutine is called. If the *attr* parameter is NULL, the default attributes are used.

The *attr* attributes object can be manipulated through a set of subroutines described in the *posix_trace_attr* family of subroutines. If the attributes of the object pointed to by the *attr* parameter are modified later, the attributes of the trace stream are not affected.

The creation-time attribute of the newly created trace stream is set to the value of the CLOCK_REALTIME clock.

The *pid* parameter represents the target process to be traced. If the *pid* parameter is zero, the calling process is traced. If the process executing this subroutine does not have appropriate privileges to trace the process identified by *pid*, an error is returned.

The `posix_trace_create` subroutine stores the trace stream identifier of the new trace stream in the object pointed to by the *trid* parameter. This trace stream identifier can be used in subsequent calls to control tracing. The *trid* parameter is used only by the following subroutines:

- **posix_trace_clear**
- **posix_trace_eventid_equal**
- **posix_trace_eventid_get_name**
- **posix_trace_eventtypelist_getnext_id**
- **posix_trace_eventtypelist_rewind**
- **posix_trace_get_attr**
- **posix_trace_get_filter**
- **posix_trace_get_status**
- **posix_trace_getnext_event**
- **posix_trace_set_filter**
- **posix_trace_shutdown**
- **posix_trace_start**

- **posix_trace_stop**
- **posix_trace_timedgetnext_event**
- **posix_trace_trid_eventid_open**
- **posix_trace_trygetnext_event**

Notice that the operations normally used by a trace analyzer process, such as the `posix_trace_rewind` or `posix_trace_close` subroutines, cannot be invoked using the trace stream identifier returned by the `posix_trace_create` subroutine.

A trace stream is created in a suspended state with an empty trace event type filter.

The `posix_trace_create` subroutine can be called multiple times from the same or different processes, with the system-wide limit indicated by the runtime invariant value *TRACE_SYS_MAX*, which has the minimum value *_POSIX_TRACE_SYS_MAX*.

The trace stream identifier returned by the `posix_trace_create` subroutine in the parameter pointed to by the *trid* parameter is valid only in the process that made the subroutine call. If it is used from another process, that is a child process, in subroutines defined in the IEEE Standard 1003.1-2001, these subroutines return with the `EINVAL` error.

If the `posix_trace_create` subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
pid	Specifies the process ID of the traced process.
attr	Specifies the trace attributes object.
trid	Specifies the trace stream identifier.

Return Values

Upon successful completion, this subroutine returns a value of zero and stores the trace stream identifier value in the object pointed to by the *trid* parameter. Otherwise, it returns the corresponding error number.

Errors

Item	Description
`EAGAIN`	No more trace streams can be started now. The value of the TRACE_SYS_MAX has been exceeded.
`EINVAL`	The *attr* parameter is null or the other parameters are invalid.
`ENOMEM`	No sufficient memory to create the trace stream with the specified parameters.
`EPERM`	Does not have appropriate privilege to trace the process specified by the *pid* parameter.
`ESRCH`	The *pid* parameter does not refer to an existing process.

Files

The **trace.h** and **types.h** files in the *Files Reference*

posix_trace_create_withlog Subroutine

Purpose

Creates an active trace stream and associates it with a trace log.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_create_withlog (pid, attr, file_desc, trid)
pid_t pid;
const trace_attr_t *restrict attr;
int file_desc;
trace_id_t *restrict trid;
```

Description

The `posix_trace_create_withlog` subroutine creates an active trace stream, as the `posix_trace_create` subroutine does, and associates the stream with a trace log.

The *file_desc* parameter must be the file descriptor designating the trace log destination. The subroutine fails if this file descriptor refers to a file opened with the *O_APPEND* flag or if this file descriptor refers to a file that is not regular.

The *trid* parameter points to the parameter where the `posix_trace_create_withlog` subroutine returns the trace stream identifier, which uniquely identifies the newly created trace stream. The trace stream identifier can be used in subsequent calls to control tracing. The *trid* parameter is only used by the following subroutines:

- `posix_trace_clear`
- `posix_trace_eventid_equal`
- `posix_trace_eventid_get_name`
- `posix_trace_eventtypelist_getnext_id`
- `posix_trace_eventtypelist_rewind`
- `posix_trace_flush`
- `posix_trace_get_attr`
- `posix_trace_get_filter`
- `posix_trace_get_status`
- `posix_trace_set_filter`
- `posix_trace_shutdown`
- `posix_trace_start`
- `posix_trace_stop`
- `posix_trace_trid_eventid_open`

Notice that the operations used by a trace analyzer process, such as the `posix_trace_rewind` or `posix_trace_close` subroutines, cannot be invoked using the trace stream identifier that is returned by the `posix_trace_create_withlog` subroutine.

For an active trace stream with log, when the `posix_trace_shutdown` subroutine is called, all trace events that have not been flushed to the trace log are flushed, as in the `posix_trace_flush` subroutine, and the trace log is closed.

When a trace log is closed, all the information that can be retrieved later from the trace log through the trace interface are written to the trace log. This information includes the trace attributes, the list of trace event types (with the mapping between trace event names and trace event type identifiers), and the trace status.

If the `posix_trace_create_withlog` subroutine is called with a non-initialized attributes object as parameter, the result is not specified.

Parameters

Item	Description
pid	Specifies the process ID of the traced process.
attr	Specifies the trace attributes object.
file_desc	Specifies the open file descriptor of the trace log.
trid	Specifies the trace stream identifier.

Return Values

Upon successful completion, this subroutine returns a value of zero and stores the trace stream identifier value in the object pointed to by the *trid* parameter. Otherwise, it returns the corresponding error number.

Errors

Item	Description
EAGAIN	No more trace streams can be started now. The value of the TRACE_SYS_MAX has been exceeded.
EBADF	The *file_desc* parameter is not a valid file descriptor open for writing.
EINVAL	The *attr* parameter is null or the other parameters are invalid. The *file_desc* parameter refers to a file with a file type that does not support the log policy associated with the trace log.
ENOMEM	No sufficient memory to create the trace stream with the specified parameters.
ENOSPC	No space left on device. The device corresponding to the *file_desc* parameter does not contain the space required to create this trace log.
EPERM	Does not have appropriate privilege to trace the process specified by the *pid* parameter.
ESRCH	The *pid* parameter does not refer to an existing process.

Files

The **trace.h** and **types.h** files in the *Files Reference*

posix_trace_event Subroutine

Purpose

Trace subroutines for implementing a trace point.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <sys/type.h>
#include <trace.h>

void posix_trace_event(event_id, data_ptr, data_len)
trace_event_id_t event_id;
const void *restrict data_ptr;
size_t data_len;
```

Description

In the trace stream that calling process is being traced and the *event_id* is not filtered out, the `posix_trace_event` subroutine records the values of the *event_id* parameter and the user data, which is specified by the *data_ptr* parameter.

The *data_len* parameter represents the total size of the user trace event data. If the value of the *data_len* is not larger than the declared maximum size for user trace event data, the *truncation-status* attribute of the trace event recorded is `POSIX_TRACE_NOT_TRUNCATED`. Otherwise, the user trace event data is truncated to this declared maximum size and the *truncation-status* attribute of the trace event recorded is `POSIX_TRACE_TRUNCATED_RECORD`.

The `posix_trace_event` subroutine has no effect in the following situations:

- No trace stream is created for the process.
- The created trace stream is not running.
- The trace event specified by the *event_id* parameter is filtered out in the trace stream.

Parameter

Item	Description
event_id	Specifies the trace event identifier.
data_ptr	Specifies the user data to be written in the trace streams that the process is tracing in.
data_len	Specifies the length of the user data to be written.

Return Values

No return value is defined for the `posix_trace_event` subroutine.

Errors

This subroutine returns no error code when it fails.

Files

The **trace.h** and **types.h** files in *Files Reference*

posix_trace_eventset_add Subroutine

Purpose

Adds a trace event type in a trace event type set.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_eventset_add (event_id, set)
trace_event_id_t event_id;
trace_event_set_t *set;
```

Description

This subroutine manipulates sets of trace event types. It operates on data objects addressable by the application, not on the current trace event filter of any trace stream.

The `posix_trace_eventset_add` subroutine adds the individual trace event type specified by the value of the *event_id* parameter to the trace event type set pointed to by the *set* parameter. Adding a trace event type already in the set is not considered as an error.

Applications call either the `posix_trace_eventset_empty` or `posix_trace_eventset_fill` subroutine at least once for each object of the *trace_event_set_t* type before further use of that object. If an object is not initialized in this way, but is supplied as a parameter to any of the `posix_trace_eventset_add`, `posix_trace_eventset_del`, or `posix_trace_eventset_ismember` subroutines, the results are not defined.

Parameters

Item	Description
eventid	Specifies the trace event identifier.
set	Specifies the set of trace event types.

Return Values

On successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following value is returned:

Item	Description
`EINVAL`	The value of one of the parameters is not valid.

Files

The `trace.h` file in the *Files Reference*

posix_trace_eventset_del Subroutine

Purpose

Deletes a trace event type from a trace event type set.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_eventset_del(event_id, set)
trace_event_id_t event_id;
trace_event_set_t *set;
```

Description

This subroutine manipulates sets of trace event types. It operates on data objects addressable by the application, not on the current trace event filter of any trace stream.

The `posix_trace_eventset_del` subroutine deletes the individual trace event type specified by the value of the *event_id* parameter from the trace event type set pointed to by the *set* argument.

Applications call either the `posix_trace_eventset_empty` or `posix_trace_eventset_fill` subroutine at least once for each object of the *trace_event_set_t* type before further use of that object. If an object is not initialized in this way, but is supplied as a parameter to any of the `posix_trace_eventset_add`, `posix_trace_eventset_del`, or `posix_trace_eventset_ismember` subroutines, the results are not defined.

Parameters

Item	Description
eventid	Specifies the trace event identifier.
set	Specifies the set of trace event types.

Return Values

On successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following value is returned:

Item	Description
`EINVAL`	The value of one of the parameters is not valid.

Files

The trace.h file in *Files Reference*.

posix_trace_eventset_empty Subroutine

Purpose

Empties a trace event type set.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_eventset_empty(set)
trace_event_set_t *set;
```

Description

This subroutine manipulates sets of trace event types. It operates on data objects addressable by the application, not on the current trace event filter of any trace stream.

The `posix_trace_eventset_empty` subroutine initializes the trace event type set pointed to by the *set* parameter so that all trace event types defined, both system and user, are excluded from the set.

Applications call either the `posix_trace_eventset_empty` or `posix_trace_eventset_fill` subroutine at least once for each object of the *trace_event_set_t* type before further use of that object. If an object is not initialized in this way, but is supplied as a parameter to any of the `posix_trace_eventset_add`, `posix_trace_eventset_del`, or `posix_trace_eventset_ismember` subroutines, the results are not defined.

Parameters

Item	Description
set	Specifies the set of trace event types.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following value is returned:

Item	Description
`EINVAL`	The value of one of the parameters is not valid.

Files

The **trace.h** file in *Files Reference*.

posix_trace_eventset_fill Subroutine

Purpose

Fills in a trace event type set.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_eventset_fill(set, what)
```

```
trace_event_set_t *set;
int what;
```

Description

This subroutine manipulates sets of trace event types. It operates on data objects addressable by the application, not on the current trace event filter of any trace stream.

The `posix_trace_eventset_fill` subroutine initializes the trace event type set pointed to by the *set* parameter. The value of the *what* parameter consists of one of the following values, as defined in the `trace.h` header file:

Item	Description
POSIX_TRACE_WOPID_EVENTS	All the system trace event types that are independent of process are included in the set.
POSIX_TRACE_SYSTEM_EVENTS	All the system trace event types are included in the set.
POSIX_TRACE_ALL_EVENTS	All trace event types that are defined, both system and user, are included in the set.

Applications call either the `posix_trace_eventset_empty` or `posix_trace_eventset_fill` subroutine at least once for each object of the *trace_event_set_t* type before further use of that object. If an object is not initialized in this way, but is supplied as a parameter to any of the `posix_trace_eventset_add`, `posix_trace_eventset_del`, or `posix_trace_eventset_ismember` subroutines, the results are not defined.

Parameters

Item	Description
set	Specifies the set of trace event types.
what	The *what* parameter contains one of the following values: **POSIX_TRACE_WOPID_EVENTS** All the system trace event types that are independent of process are included in the set. **POSIX_TRACE_SYSTEM_EVENTS** All the system trace event types are included in the set. **POSIX_TRACE_ALL_EVENTS** All trace event types that are defined, both system and user, are included in the set.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following value is returned:

Item	Description
`EINVAL`	The value of one of the parameters is not valid.

Files

The **trace.h** file in *Files Reference*.

posix_trace_eventset_ismember Subroutine

Purpose

Tests if the trace event type is included in the trace event type set.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_eventset_ismember(event_id, set, ismember)
trace_event_id_t event_id;
const trace_event_set_t *restrict set;
int *restrict ismember;
```

Description

This subroutine manipulates sets of trace event types. It operates on data objects addressable by the application, not on the current trace event filter of any trace stream.

Applications call either the `posix_trace_eventset_empty` or `posix_trace_eventset_fill` subroutine at least once for each object of the *trace_event_set_t* type before further use of that object. If an object is not initialized in this way, but is supplied as a parameter to any of the `posix_trace_eventset_add`, `posix_trace_eventset_del`, or `posix_trace_eventset_ismember` subroutines, the results are undefined.

The `posix_trace_eventset_ismember` subroutine tests whether the trace event type specified by the value of the *event_id* parameter is a member of the set pointed to by the *set* parameter. The value returned in the object pointed to by the *ismember* parameter is zero if the trace event type identifier is not a member of the set. It returns a nonzero value if it is a member of the set.

Parameters

Item	Description
eventid	Specifies the trace event identifier.
set	Specifies the set of trace event types.
ismember	Specifies the returned value of the `posix_trace_eventset_ismember` subroutine.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following value is returned:

Item	Description
`EINVAL`	The value of one of the parameters is not valid.

Files

The **trace.h** file in *Files Reference*.

posix_trace_eventid_equal Subroutine

Purpose

Compares two trace event type identifiers.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_eventid_equal(trid, event1, event2)
trace_id_t trid;
trace_event_id_t event1;
trace_event_id_t event2;
```

Description

The `posix_trace_eventid_equal` compares the *event1* and *event2* trace event type identifiers. If the *event1* and *event2* identifiers are equal (from the same trace stream, the same trace log or from different trace streams), the return value is non-zero; otherwise, a value of zero is returned.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
event, *event1*, *event2*	Specifies the trace event identifiers.

Return Values

The `posix_trace_eventid_equal` subroutine returns a non-zero value if the value of the *event1* and *event2* parameters are equal; otherwise, a value of zero is returned.

Error

This subroutine returns no error code.

File

The **trace.h** file in *Files Reference*

posix_trace_eventid_open Subroutine

Purpose

Trace subroutine for instrumenting application code.

Library

Posix Trace Library (libposixtrace.a)

Syntax

```
#include <sys/type.h>
#include <trace.h>

int posix_trace_eventid_open(event_name, event_id)
const char *restrict event_name;
trace_event_id_t *restrict event_id;
```

Description

The posix_trace_eventid_open subroutine associates a user trace event name with a trace event type identifier for the calling process. The trace event name is the string pointed to by the *event_name* parameter. It can have a maximum number of characters defined in the *TRACE_EVENT_NAME_MAX* (which has the minimum value of *_POSIX_TRACE_EVENT_NAME_MAX*). The number of user trace event type identifiers that can be defined for any given process is limited by the maximum value defined in the *TRACE_USER_EVENT_MAX*, which has the minimum value *_POSIX_TRACE_USER_EVENT_MAX*.

The posix_trace_eventid_open subroutine associates the user trace event name pointed to by the *event_name* parameter with a trace event type identifier that is unique for all of the processes being traced in this same trace stream, and is returned in the variable pointed to by the *event_id* parameter. If the user trace event name has already been mapped for the traced processes, the previously assigned trace event type identifier is returned. If the per-process user trace event name limit represented by the *TRACE_USER_EVENT_MAX* value has been reached, the pre-defined *POSIX_TRACE_UNNAMED_USEREVENT* user trace event is returned.

Note: The above procedure, together with the fact that multiple processes can only be traced into the same trace stream by inheritance, ensure that all the processes that are traced into a trace stream have the same mapping of trace event names to trace event type identifiers.

If there is no trace stream created, the posix_trace_eventid_open subroutine stores this information for future trace streams created for this process.

Parameter

Item	Description
event_name	Specifies the trace event name.
event_id	Specifies the trace event identifier.

Return Values

On successful completion, the posix_trace_eventid_open subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the posix_trace_eventid_open subroutine stores the trace event type identifier value in the object pointed to by *event_id*.

Errors

The posix_trace_eventid_open subroutine fails if the following error returns:

Item	Description
ENAMETOOLONG	The size of the name pointed to by the *event_name* parameter is longer than the value defined by *TRACE_EVENT_NAME_MAX*.

Files

The **trace.h** and **types.h** files in *Files Reference*.

posix_trace_eventid_get_name Subroutine

Purpose

Retrieves the trace event name from a trace event type identifier.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_eventid_get_name(trid, event, event_name)
trace_id_t trid;
trace_event_id_t event;
char *event_name;
```

Description

The `posix_trace_eventid_get_name` subroutine returns the trace event name associated with the trace event type identifier for a trace stream or a trace log in the argument pointed to by the *event_name* parameter. The *event* argument defines the trace event type identifier. The *trid* argument defines the trace stream or the trace log. The name of the trace event will have a maximum number of characters defined in the *TRACE_EVENT_NAME_MAX* variable, which has the minimum value _POSIX_TRACE_EVENT_NAME_MAX. Successive calls to this subroutine with the same trace event type identifier and the same trace stream identifier return the same event name.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
event	Specifies the trace event identifier.
event_name	Specifies the trace event name.

Return Values

On successful completion, the `posix_trace_eventid_get_name` subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_eventid_get_name` subroutine stores the trace event name value in the object pointed to by the *event_name* parameter.

Errors

The `posix_trace_eventid_get_name` subroutine fails if the following value returns:

Item	Description
EINVAL	The *trid* argument is not a valid trace stream identifier. The trace event type identifier event is not associated with any name.

File

The `trace.h` file in *Files Reference*.

posix_trace_eventtypelist_getnext_id and posix_trace_eventtypelist_rewind Subroutines

Purpose

Iterate over a mapping of trace event types.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_eventtypelist_getnext_id(trid, event, unavailable)
trace_id_t trid;
trace_event_id_t *restrict event;
int *restrict unavailable;

int posix_trace_eventtypelist_rewind(trid)
trace_id_t trid;
```

Description

The first time the `posix_trace_eventtypelist_getnext_id` subroutine is called, it returns the first trace event type identifier of the list of trace events identified by the *trid* parameter. The identifier is returned in the *event* variable. The trace events belong to the trace stream that is identified by the *trid* parameter. Successive calls to the `posix_trace_eventtypelist_getnext_id` subroutine return in the *event* variable the next trace event type identifier in that same list. Each time a trace event type identifier is successfully written into the *event* parameter, the *unavailable* parameter is set to zero. When no more trace event type identifiers are available, the *unavailable* parameter is set to a value of nonzero.

The `posix_trace_eventtypelist_rewind` subroutine resets the next trace event type identifier, so it is read to the first trace event type identifier from the list of trace events that is used in the trace stream identified by the *trid* parameter.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
event	Specifies the trace event identifier.
unavailable	Specifies the location set to zero if a trace event type is reported; otherwise, it is nonzero.

Return Values

On successful completion, these subroutines return a value of zero. Otherwise, they return the corresponding error number.

If successful, the `posix_trace_eventtypelist_getnext_id` subroutine stores the trace event type identifier value in the object pointed to by the *event* parameter.

Errors

These subroutines fail if the following value returns:

Item	Description
EINVAL	The *trid* parameter is not a valid trace stream identifier.

Files

The **trace.h** file in *Files Reference*.

posix_trace_flush Subroutine

Purpose

Initiates a flush on the trace stream.

Library

Posix Trace Library (libposixtrace.a)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_flush (trid)
trace_id_t trid;
```

Description

The posix_trace_flush subroutine initiates a flush operation that copies the contents of the trace stream identified by the *trid* parameter into the trace log associated with the trace stream at the creation time. If no trace log has been associated with the trace stream pointed to by the *trid* parameter, this subroutine returns an error. The termination of the flush operation can be polled by the **posix_trace_get_status** subroutine. After the flushing is completed, the space used by the flushed trace events is available for tracing new trace events. During the flushing operation, it is possible to trace new trace events until the trace stream becomes full.

If flushing the trace stream makes the trace log full, the trace log full policy is applied. If the trace log-full-policy attribute is set, the following occurs:

POSIX_TRACE_UNTIL_FULL
The trace events that have not been flushed are discarded.

POSIX_TRACE_LOOP
The trace events that have not been flushed are written to the beginning of the trace log, overwriting previous trace events stored there.

POSIX_TRACE_APPEND
The trace events that have not been flushed are appended to the trace log.

For an active trace stream with the log, when the posix_trace_shutdown subroutine is called, all trace events that have not been flushed to the trace log are flushed, and the trace log is closed.

When a trace log is closed, all the information that can be retrieved later from the trace log through the trace interface are written to the trace log. This information includes the trace attributes, the list of trace event types (with the mapping between trace event names and trace event type identifiers), and the trace status.

The posix_trace_shutdown subroutine does not return until all trace events have been flushed.

Parameters

Item	Description
trid	Specifies the trace stream identifier.

Return Values

On successful completion, these subroutines return a value of zero. Otherwise, they return the corresponding error number.

Errors

Item	Description
EINVAL	The value of the *trid* parameter does not correspond to an active trace stream with log.
ENOSPC	No space left on device.

Files

The **trace.h** and the **types.h** files in *Files Reference*.

posix_trace_getnext_event Subroutine

Purpose

Retrieves a trace event.

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_getnext_event(trid, event, data, num_bytes, data_len, unavailable)
trace_id_t trid;
struct posix_trace_event_info *restrict event;
void *restrict data;
size_t num_bytes;
size_t *restrict data_len;
int *restrict unavailable;
```

Description

The **posix_trace_getnext_event** subroutine reports a recorded trace event either from an active trace stream without a log or a pre-recorded trace stream identified by the *trid* parameter.

The trace event information associated with the recorded trace event is copied by the function into the structure pointed to by the *event* parameter, and the data associated with the trace event is copied into the buffer pointed to by the *data* parameter.

The **posix_trace_getnext_event** subroutine blocks if the *trid* parameter identifies an active trace stream and there is currently no trace event ready to be retrieved. When returning, if a recorded trace event was reported, the variable pointed to by the *unavailable* parameter is set to 0. Otherwise, the variable pointed to by the *unavailable* parameter is set to a value different from 0.

The *num_bytes* parameter equals the size of the buffer pointed to by the *data* parameter. The *data_len* parameter reports to the application the length, in bytes, of the data record just transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace event pointed to by the *event* parameter, all the recorded data is transferred. In this case, the truncation-status member of the trace event structure is either POSIX_TRACE_NOT_TRUNCATED (if the trace event data was recorded without

truncation while tracing) or POSIX_TRACE_TRUNCATED_RECORD (if the trace event data was truncated when it was recorded). If the *num_bytes* parameter is less than the length of the recorded trace event data, the data transferred is truncated to the length of *num_bytes*, that is the value stored in the variable pointed to by *data_len* equals *num_bytes*, and the truncation-status member of the *event* structure parameter is set to POSIX_TRACE_TRUNCATED_READ (see the `posix_trace_event_info` structure defined in **trace.h**).

The report of a trace event is sequential starting from the oldest recorded trace event. Trace events are reported in the order in which they were generated, up to an implementation-defined time resolution that causes the ordering of trace events to be occurring very close to each other to be unknown. After it is reported, a trace event cannot be reported again from an active trace stream. After a trace event is reported from an active trace stream without the log, the trace stream makes the resources associated with that trace event available to record future generated trace events.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
event	Specifies the `posix_trace_event_info` structure that contains the trace event information of the recorded event.
data	Specifies the user data associated with the trace event.
num_bytes	Specifies the size, in bytes, of the buffer pointed to by the data parameter.
data_len	Specifies the size, in bytes, of the user data record just transferred.
unavailable	Specifies the location set to 0 if an event is reported. Otherwise, specifies a value of nonzero.

Return Values

On successful completion, the **posix_trace_getnext_event** subroutine returns a value of 0. Otherwise, it returns the corresponding error number.

If successful, the **posix_trace_getnext_event** subroutine stores:

- The recorded trace event in the object pointed to by *event*
- The trace event information associated with the recorded trace event in the object pointed to by *data*
- The length of this trace event information in the object pointed to by *data_len*
- The value of 0 in the object pointed to by *unavailable*

Error Codes

the **posix_trace_getnext_event** subroutine fails if the following error codes return:

Item	Description
`EINVAL`	The trace stream identifier parameter *trid* is not valid.
`EINTR`	The operation was interrupted by a signal, and so the call had no effect.

Files

The **pthread.h**, **trace.h** and **types.h** in *Files Reference*.

posix_trace_get_attr Subroutine

Purpose

Retrieve trace attributes.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_get_attr(trid, attr)
trace_id_t trid;
trace_attr_t *attr;
```

Description

The **posix_trace_get_attr** subroutine copies the attributes of the active trace stream identified by the *trid* into the *attr* parameter. The *trid* parameter might represent a pre-recorded trace log.

If the **posix_trace_get_attr** subroutine is called with a non-initialized attribute object as a parameter, the result is not specified.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
attr	Specifies the trace attributes object.

Return Values

On successful completion, the `posix_trace_get_attr` subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_get_attr` subroutine stores the trace attributes in the *attr* parameter.

Errors

The `posix_trace_get_attr` subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The *trid* trace stream parameter does not correspond to a valid active trace stream or a valid trace log.

Files

The **trace.h** file in the *Files Reference*.

posix_trace_get_filter Subroutine

Purpose

Retrieves the filter of an initialized trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_get_filter(trid, set)
trace_id_t trid;
trace_event_set_t *set;
```

Description

The `posix_trace_get_filter` subroutine retrieves into the *set* parameter the actual trace event filter from the trace stream specified by the *trid* parameter.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
set	Points to the set of trace event types.

Return Values

On successful completion, the `posix_trace_get_filter` subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_get_filter` subroutine stores the set of filtered trace event types in the *set* parameter.

Errors

It fails if the following value returns:

Item	Description
`EINVAL`	The value of the *trid* parameter does not correspond to an active trace stream or the value of the parameter pointed to by the *set* parameter is not valid.

Files

The **trace.h** file in *Files Reference*.

posix_trace_get_status Subroutine

Purpose

Retrieves trace attributes or trace status.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>
int posix_trace_get_status(trid, statusinfo)
trace_id_t trid;
struct posix_trace_status_info *statusinfo;
```

Description

The `posix_trace_get_status` subroutine returns, in the structure pointed to by the *statusinfo* parameter, the current trace status for the trace stream identified by the *trid* parameter. If the *trid* parameter refers to a pre-recorded trace stream, the *status* parameter is the status of the completed trace stream.

When the **posix_trace_get_status** subroutine is used, the *overrun* status of the trace stream is reset to the *POSIX_TRACE_NO_OVERRUN* value after the call completes. See the **trace.h** File for further information.

If the *trid* parameter refers to a trace stream with a log, when the **posix_trace_get_status** subroutine is used, the log's *overrun* status of the trace stream is reset to the *POSIX_TRACE_NO_OVERRUN* value and the *flush_error* status is reset to a value of zero after the call completes.

If the *trid* parameter refers to a pre-recorded trace stream, the status that is returned is the status of the completed trace stream and the status values of the trace stream are not reset.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
statusinfo	Specifies the current trace status.

Return Values

On successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_get_status` subroutine stores the trace status in the *statusinfo* parameter.

Errors

The `posix_trace_get_status` subroutine fails if the following error number returns:

Item	Description
`EINVAL`	The *trid* trace stream parameter does not correspond to a valid active trace stream or a valid trace log.

Files

The **trace.h** file in the *Files Reference*.

posix_trace_open Subroutine

Purpose

Opens a trace log.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_open (file_desc, trid)
int file_desc;
trace_id_t *trid;
```

Description

The `posix_trace_open` subroutine allocates the necessary resources and establish the connection between a trace log identified by the *file_desc* parameter and a trace stream identifier identified by the object pointed to by the *trid* parameter. The *file_desc* parameter must be a valid open file descriptor that corresponds to a trace log. The *file_desc* parameter must be open for reading. The current trace event time stamp is set to the time stamp of the oldest trace event recorded in the trace log identified by the *trid* parameter. The current trace event time stamp specifies the time stamp of the trace event that will be read by the next call to the `posix_trace_getnext_event`.

The `posix_trace_open` subroutine returns a trace stream identifier in the variable pointed to by the *trid* parameter, which might only be used by the following subroutines:

- The `posix_trace_close` subroutine
- The `posix_trace_eventid_equal` subroutine
- The `posix_trace_eventid_get_name` subroutine
- The `posix_trace_eventtypelist_getnext_id` subroutine
- The `posix_trace_eventtypelist_rewind` subroutine
- The `posix_trace_get_attr` subroutine
- The `posix_trace_get_status` subroutine
- The `posix_trace_getnext_event` subroutine
- The `posix_trace_rewind` subroutine

Note that the operations used by a trace controller process, such as the `posix_trace_start`, `posix_trace_stop`, or the `posix_trace_shutdown` subroutine, cannot be invoked using the trace stream identifier returned by the `posix_trace_open` subroutine.

Parameters

Item	Description
file_desc	Specifies the open file descriptor of the trace log.
trid	Specifies the trace stream identifier.

Return Values

On successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_open` subroutine stores the trace stream identifier value in the object pointed to by the *trid* parameter.

Errors

The `posix_trace_open` subroutine fails if the following errors return:

Item	Description
EBADF	The *file_desc* parameter is not a valid file descriptor open for reading.
EINVAL	The object pointed to by *file_desc* does not correspond to a valid trace log.

Files

The **trace.h** file in the *Files Reference*.

posix_trace_rewind Subroutine

Purpose

Re-initializes the trace log for reading.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_rewind (trid)
trace_id_t trid;
```

Description

The `posix_trace_rewind` subroutine resets the current trace event time stamp to the time stamp of the oldest trace event recorded in the trace log identified by the *trid* parameter. The current trace event time stamp specifies the time stamp of the trace event that will be read by the next call to `posix_trace_getnext_event` subroutine.

Parameters

Item	Description
trid	Specifies the trace stream identifier.

Return Values

On successful completion, the subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The `posix_trace_rewind` subroutine fails if the following error returns:

Item	Description
EINVAL	The object pointed to by the *trid* parameter does not correspond to a valid trace log.

Files

The **trace.h** file in the *Files Reference*.

posix_trace_set_filter Subroutine

Purpose

Sets the filter of an initialized trace stream.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_set_filter(trid, set, how)
trace_id_t trid;
const trace_event_set_t *set;
int how;
```

Description

The `posix_trace_set_filter` subroutine changes the set of filtered trace event types after a trace stream identified by the *trid* parameter is created. This subroutine can be called before starting the trace stream, or while the trace stream is active. By default, if no call is made to the `posix_trace_set_filter`, all trace events are recorded (that is, none of the trace event types is filtered out).

If this subroutine is called while the trace is in progress, a special system trace event, the POSIX_TRACE_FILTER, is recorded in the trace indicating both the old and the new sets of filtered trace event types. The POSIX_TRACE_FILTER is a System Trace Event type associated with a trace event type filter change operation.

The *how* parameter indicates the way that the *set* parameter is to be changed. It has one of the following values, as defined in the **trace.h** header:

POSIX_TRACE_SET_EVENTSET
The set of trace event types to be filtered is the trace event type set that the *set* parameter points to.

POSIX_TRACE_ADD_EVENTSET
The set of trace event types to be filtered is the union of the current set and the trace event type set that the *set* parameter points to.

POSIX_TRACE_SUB_EVENTSET
The set of trace event types to be filtered is all trace event types in the current set that are not in the set that the *set* parameter points to; that is, remove each element of the specified set from the current filter.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
set	Points to the set of trace event types.
how	Specifies the operation to be done on the set.

Return Values

On successful completion, it returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

This subroutine fails if the following value returns:

Item	Description
EINVAL	The value of the *trid* parameter does not correspond to an active trace stream or the value of the parameter pointed to by the *set* parameter is not valid.

Files

The **trace.h** file in *Files Reference*.

posix_trace_shutdown Subroutine

Purpose

Shuts down a trace stream.

Library

Posix Trace Library (libposixtrace.a)

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_shutdown (trid)
trace_id_t trid;
```

Description

The posix_trace_shutdown subroutine stops the tracing of trace events in the trace stream identified by the *trid* parameter, as if the posix_trace_stop subroutine had been invoked. The posix_trace_shutdown subroutine frees all the resources associated with the trace stream.

The posix_trace_shutdown subroutine does not return until all the resources associated with the trace stream have been freed. When the posix_trace_shutdown subroutine returns, the *trid* parameter becomes an invalid trace stream identifier. A call to this subroutine deallocates the resources regardless of whether all trace events have been retrieved by the analyzer process. Any thread blocked on the posix_trace_getnext_event, posix_trace_timedgetnext_event or the posix_trace_trygetnext_event subroutines before this call is unblocked and the EINVAL error is returned.

The trace streams are automatically shut down when the processes that create them start any subroutines of the **exec** subroutines, or when the processes are terminated.

For an active trace stream with log, when the posix_trace_shutdown subroutine is called, all trace events that have not been flushed to the trace log are flushed, as in the posix_trace_flush subroutine, and the trace log is closed.

When a trace log is closed, all the information that can be retrieved later from the trace log through the trace interface are written to the trace log. This information includes the trace attributes, the list of trace event types (with the mapping between trace event names and trace event type identifiers), and the trace status.

The posix_trace_shutdown subroutine does not return until all trace events have been flushed.

Parameters

Item	Description
trid	Specifies the trace stream identifier.

Return Values

Upon successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

Item	Description
EINVAL	The value of the *trid* parameter does not correspond to an active trace stream with log.
ENOSPC	No space left on device.

Files

The trace.h and types.h files in *Files Reference*

posix_trace_start Subroutine

Purpose

Starts a trace.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_start(trid)
trace_id_t trid;
```

Description

The `posix_trace_start` subroutine starts the trace stream identified by the *trid* parameter.

The effect of calling the `posix_trace_start` subroutine is recorded in the trace stream as the *POSIX_TRACE_START* system trace event, and the status of the trace stream becomes *POSIX_TRACE_RUNNING*. If the trace stream is in progress when this subroutine is called, the *POSIX_TRACE_START* system trace event is not recorded, and the trace stream continues to run. If the trace stream is full, the *POSIX_TRACE_START* system trace event is not recorded, and the status of the trace stream is not changed.

Parameters

Item	Description
trid	Specifies the trace stream identifier.

Return Values

On successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The subroutine fails if the following error number returns:

Item	Description
EINVAL	The value of the *trid* parameter does not correspond to an active trace stream and thus no trace stream is started or stopped.

Files

The **trace.h** file in *Files Reference*.

posix_trace_stop Subroutine

Purpose

Stops a trace.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_stop(trid)
trace_id_t trid;
```

Description

The `posix_trace_stop` subroutine stops the trace stream identified by the *trid* parameter.

The effect of calling the `posix_trace_stop` subroutine is recorded in the trace stream as the *POSIX_TRACE_STOP* system trace event, and the status of the trace stream becomes *POSIX_TRACE_SUSPENDED*. If the trace stream is suspended when this subroutine is called, the *POSIX_TRACE_STOP* system trace event is not recorded, and the trace stream remains suspended. If the trace stream is full, the *POSIX_TRACE_STOP* system trace event is not recorded, and the status of the trace stream is not changed.

Parameters

Item	Description
trid	Specifies the trace stream identifier.

Return Values

On successful completion, this subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

Errors

The subroutine fails if the following error number returns:

Item	Description
EINVAL	The value of the *trid* parameter does not correspond to an active trace stream and thus no trace stream is started or stopped.

Files

The **trace.h** file in *Files Reference*.

posix_trace_timedgetnext_event Subroutine

Purpose

Retrieves a trace event.

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_timedgetnext_event
(trid, event, data, num_bytes, data_len, unavailable, abs_timeout)
trace_id_t trid;
struct posix_trace_event_info *restrict event;
void *restrict data;
size_t num_bytes;
size_t *restrict data_len;
int *restrict unavailable;
const struct timespec *restrict abs_timeout;
```

Description

The **posix_trace_timedgetnext_event** subroutine attempts to get another trace event from an active trace stream without a log, as in the **posix_trace_getnext_event** subroutine. However, if no trace event is available from the trace stream, the implied wait terminates when the timeout specified by the parameter *abs_timeout* expires, and the function returns the error [ETIMEDOUT].

The timeout expires when the absolute time specified by *abs_timeout* passes or has already passed at the time of the call. The absolute time specified by the *abs_timeout* is measured by the clock on which a timeout is based (that is, when the value of that clock equals or exceeds *abs_timeout*).

The timeout is based on the CLOCK_REALTIME clock. The resolution of the timeout is the resolution of the CLOCK_REALTIME. The `timespec` data type is defined in the `time.h` header file.

The function never fails with a timeout if a trace event is immediately available from the trace stream. The validity of the *abs_timeout* parameter is not checked if a trace event is immediately available from the trace stream.

The behavior of this subroutine for a pre-recorded trace stream is not specified.

The *num_bytes* parameter equals the size of the buffer pointed to by the *data* parameter. The *data_len* parameter reports to the application the length, in bytes, of the data record just transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace event pointed to by the *event* parameter, all the recorded data is transferred. In this case, the truncation-status member of the trace event structure is either POSIX_TRACE_NOT_TRUNCATED (if the trace event data was recorded without truncation while tracing) or POSIX_TRACE_TRUNCATED_RECORD (if the trace event data was truncated when it was recorded). If the *num_bytes* parameter is less than the length of the recorded trace event data, the data transferred is truncated to the length of the *num_bytes* parameter, the value stored in the variable pointed to by *data_len* equals *num_bytes*, and the truncation-status member of the *event*

structure parameter is set to POSIX_TRACE_TRUNCATED_READ (see the `posix_trace_event_info` structure defined in **trace.h**).

The report of a trace event is sequential starting from the oldest recorded trace event. Trace events are reported in the order in which they were generated, up to an implementation-defined time resolution that causes the ordering of trace events occurring very close to each other to be unknown. After it is reported, a trace event cannot be reported again from an active trace stream. After a trace event is reported from an active trace stream without a log, the trace stream makes the resources associated with that trace event available to record future generated trace events.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
event	Specifies the `posix_trace_event_info` structure that contains the trace event information of the recorded event.
data	Specifies the user data associated with the trace event.
num_bytes	Specifies the size, in bytes, of the buffer pointed to by the data parameter.
data_len	Specifies the size, in bytes, of the user data record just transferred.
unavailable	Specifies the location set to 0 if an event is reported, or non zero otherwise.
abs_timeout	Specifies a structure of the **timespec** type struct .

Return Values

On successful completion, the **posix_trace_timedgetnext_event** subroutine returns a value of 0. Otherwise, it returns the corresponding error number.

If successful, the **posix_trace_timedgetnext_event** subroutine stores:

- The recorded trace event in the object pointed to by *event*
- The trace event information associated with the recorded trace event in the object pointed to by *data*
- The length of this trace event information in the object pointed to by *data_len*
- The value of 0 in the object pointed to by *unavailable*

Error Codes

The **posix_trace_timedgetnext_event** subroutine fails if the following error codes return:

Item	Description
`EINVAL`	The trace stream identifier parameter *trid* is not valid.
`EINVAL`	There is no trace event immediately available from the trace stream, and the *timeout* parameter is not valid.
`EINTR`	The operation was interrupted by a signal, and so the call had no effect.
`ETIMEDOUT`	No trace event was available from the trace stream before the specified *timeout* expired.

Files

The **pthread.h**, **trace.h** and **types.h** in *Files Reference*.

posix_trace_trygetnext_event Subroutine

Purpose

Retrieves a trace event.

Syntax

```
#include <sys/types.h>
#include <trace.h>

int posix_trace_trygetnext_event(trid, event, data, num_bytes, data_len, unavailable)
trace_id_t trid;
struct posix_trace_event_info *restrict event;
void *restrict data;
size_t num_bytes;
size_t *restrict data_len;
int *restrict unavailable;
```

Description

The **posix_trace_trygetnext_event** subroutine reports a recorded trace event from an active trace stream without a log identified by the *trid* parameter.

The trace event information associated with the recorded trace event is copied by the function into the structure pointed to by the *event* parameter, and the data associated with the trace event is copied into the buffer pointed to by the *data* parameter.

The **posix_trace_trygetnext_event** subroutine does not block. This function returns an error if the *trid* parameter identifies a pre-recorded trace stream. If a recorded trace event was reported, the variable pointed to by the *unavailable* parameter is set to 0. Otherwise, if no trace event was reported, the variable pointed to by the *unavailable* parameter is set to a value different from zero.

The *num_bytes* parameter equals the size of the buffer pointed to by the *data* parameter. The *data_len* parameter reports to the application the length, in bytes, of the data record just transferred. If *num_bytes* is greater than or equal to the size of the data associated with the trace event pointed to by the *event* parameter, all the recorded data is transferred. In this case, the truncation-status member of the trace event structure is either POSIX_TRACE_NOT_TRUNCATED (if the trace event data was recorded without truncation while tracing) or POSIX_TRACE_TRUNCATED_RECORD (if the trace event data was truncated when it was recorded). If the *num_bytes* parameter is less than the length of recorded trace event data, the data transferred is truncated to a length of *num_bytes*, the value stored in the variable pointed to by *data_len* equals *num_bytes*, and the truncation-status member of the *event* structure parameter is set to POSIX_TRACE_TRUNCATED_READ (see the `posix_trace_event_info` structure defined in **trace.h**).

The report of a trace event is sequential starting from the oldest recorded trace event. Trace events are reported in the order in which they were generated, up to an implementation-defined time resolution that causes the ordering of trace events occurring very close to each other to be unknown. After it is reported, a trace event cannot be reported again from an active trace stream. After a trace event is reported from an active trace stream without a log, the trace stream makes the resources associated with that trace event available to record future generated trace events.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
event	Specifies the `posix_trace_event_info` structure that contains the trace event information of the recorded event.
data	Specifies the user data associated with the trace event.
num_bytes	Specifies the size, in bytes, of the buffer pointed to by the data parameter.

Item	Description
data_len	Specifies the size, in bytes, of the user data record just transferred.
unavailable	Specifies the location set to 0 if an event is reported. Otherwise, specifies the value of nonzero.

Return Values

On successful completion, the **posix_trace_trygetnext_event** subroutine returns a value of 0. Otherwise, it returns the corresponding error number.

If successful, the **posix_trace_trygetnext_event** subroutine stores:

- The recorded trace event in the object pointed to by *event*
- The trace event information associated with the recorded trace event in the object pointed to by *data*
- The length of this trace event information in the object pointed to by *data_len*
- The value of 0 in the object pointed to by *unavailable*

Error Codes

The **posix_trace_trygetnext_event** subroutine fails if the following error code returns:

Item	Description
EINVAL	The trace stream identifier parameter *trid* is not valid.
	The trace stream identifier parameter *trid* does not correspond to an active trace stream.

Files

The **pthread.h**, **trace.h** and **types.h** in *Files Reference*.

posix_trace_trid_eventid_open Subroutine

Purpose

Associates a trace event type identifier to a user trace event name.

Library

Posix Trace Library (`libposixtrace.a`)

Syntax

```
#include <trace.h>

int posix_trace_trid_eventid_open(trid, event_name, event)
trace_id_t trid;
const char *restrict event_name;
trace_event_id_t *restrict event;
```

Description

The `posix_trace_trid_eventid_open` subroutine associates a user trace event name with a trace event type identifier for a given trace stream. The trace stream is identified by the *trid* parameter, and it need to be an active trace stream. The *event_name* parameter points to the trace event name that is a string. It must have a maximum number of the characters that is defined in the

TRACE_EVENT_NAME_MAX variable, (which has the minimum value *_POSIX_TRACE_EVENT_NAME_MAX*.) The number of user trace event type identifiers that can be defined for any given process is limited by the maximum value defined by the *TRACE_USER_EVENT_MAX* that has the minimum value of *_POSIX_TRACE_USER_EVENT_MAX*.

The `posix_trace_trid_eventid_open` subroutine associates the user trace event name with a trace event type identifier for a given trace stream. The trace event type identifier is unique for all of the processes being traced in the trace stream. The *trid* parameter defines the trace stream. The trace event type identifier is returned in the variable pointed to by the *event* parameter. If the user trace event name is already mapped for the traced processes, the previously assigned trace event type identifier is returned. If the per-process user trace event name limit represented by the *TRACE_USER_EVENT_MAX* value is reached, the *POSIX_TRACE_UNNAMED_USEREVENT* user trace event previously defined is returned.

Parameters

Item	Description
trid	Specifies the trace stream identifier.
event_name	Specifies the trace event name.
event	Specifies the trace event identifiers.

Return Values

On successful completion, the `posix_trace_trid_eventid_open` subroutine returns a value of zero. Otherwise, it returns the corresponding error number.

If successful, the `posix_trace_trid_eventid_open` subroutine stores the value of the trace event type identifier in the object pointed to by the *event* parameter.

Errors

The `posix_trace_trid_eventid_open` subroutine fails if one of the following value returns:

Item	Description
`EINVAL`	The *trid* parameter is not a valid trace stream identifier. The trace event type identifier event is not associated with any name.
`ENAMETOOLONG`	The size of the name pointed to by the *event_name* parameter is longer than the *TRACE_EVENT_NAME_MAX.*

File

The **trace.h** file in *Files Reference*.

powf, powl, pow, powd32, powd64, and powd128 Subroutines

Purpose

Computes power.

Syntax

```
#include <math.h>

float powf (x, y)
float x;
float y;

long double powl (x, y)
```

```
long double x, y;

double pow (x, y)
double x, y;
_Decimal32 powd32 (x, y)
_Decimal32 x, y;

_Decimal64 powd64 (x, y)
_Decimal64 x, y;

_Decimal128 powd128 (x, y)
_Decimal128 x, y;
```

Description

The **powf**, **powl**, **pow**, **powd32**, **powd64**, and **powd128** subroutines compute the value of *x* raised to the power *y*, x^y. If *x* is negative, the application ensures that *y* is an integer value.

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value of the base.
y	Specifies the value of the exponent.

Return Values

Upon successful completion, the **pow**, **powf**, **powl**, **powd32**, **powd64**, and **powd128** subroutines return the value of *x* raised to the power *y*.

For finite values of $x < 0$, and finite non-integer values of *y*, a domain error occurs and a NaN is returned.

If the correct value would cause overflow, a range error occurs and the **pow**, **powf**, **powl**, **powd32**, **powd64**, and **powd128** subroutines return **HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If *x* or *y* is a NaN, a NaN is returned (unless specified elsewhere in this description).

For any value of y (including NaN), if *x* is +1, 1.0 is returned.

For any value of *x* (including NaN), if *y* is ±0, 1.0 is returned.

For any odd integer value of $y>0$, if *x* is ±0, ±0 is returned.

For $y > 0$ and not an odd integer, if *x* is ±0, +0 is returned.

If *x* is -1, and *y* is ±Inf, 1.0 is returned.

For $|x|<1$, if *y* is -Inf, +Inf is returned.

For $|x|>1$, if *y* is -Inf, +0 is returned.

For $|x|<1$, if *y* is +Inf, +0 is returned.

For $|x|>1$, if *y* is +Inf, +Inf is returned.

For *y* an odd integer < 0, if *x* is -Inf, -0 is returned.

For $y < 0$ and not an odd integer, if *x* is -Inf, +0 is returned.

For y an odd integer > 0, if *x* is -Inf, -Inf is returned.

For *y* > 0 and not an odd integer, if *x* is -Inf, +Inf is returned.

For *y* <0, if *x* is +Inf, +0 is returned.

For *y* >0, if *x* is +Inf, +Inf is returned.

For *y* an odd integer < 0, if *x* is ±0, a pole error occurs and ±**HUGE_VAL**, ±**HUGE_VALF**, ±**HUGE_VALL**, ±**HUGE_VAL_D32**, ±**HUGE_VAL_D64**, and ±**HUGE_VAL_D128** is returned for **pow**, **powf**, **powl**, **powd32**, **powd64**, and **powd128** respectively.

For *y* < 0 and not an odd integer, if *x* is ±0, a pole error occurs and **HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** is returned for **pow**, **powf**, **powl**, **powd32**, **powd64**, and **powd128** respectively.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Error Codes

When using the **libm.a** library:

Item	Description
pow	If the correct value overflows, the **pow**subroutine returns a **HUGE_VAL** value and sets **errno** to **ERANGE**. If the *x* parameter is negative and the *y* parameter is not an integer, the **pow** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If x=0 and the *y* parameter is negative, the **pow** subroutine returns a **HUGE_VAL** value but does not modify **errno.**
powl	If the correct value overflows, the **powl**subroutine returns a **HUGE_VAL** value and sets **errno** to **ERANGE**. If the *x* parameter is negative and the *y* parameter is not an integer, the **powl** subroutine returns a **NaNQ** value and sets **errno** to **EDOM**. If x=0 and the *y* parameter is negative, the **powl** subroutine returns a **HUGE_VAL** value but does not modify **errno.**

When using **libmsaa.a**(**-lmsaa**):

Item	Description
pow	If x=0 and the *y* parameter is not positive, or if the *x* parameter is negative and the *y* parameter is not an integer, the **pow** subroutine returns 0 and sets **errno** to **EDOM**. In these cases a message indicating DOMAIN error is output to standard error. When the correct value for the **pow** subroutine would overflow or underflow, the **pow** subroutine returns: **+HUGE_VAL** OR **-HUGE_VAL** OR 0 When using either the **libm.a** library or the **libsaa.a** library:
powl	If the correct value overflows, **powl** returns **HUGE_VAL** and **errno** to **ERANGE**. If *x* is negative and *y* is not an integer, **powl** returns **NaNQ** and sets **errno** to **EDOM**. If *x* = zero and *y* is negative, **powl** returns a **HUGE_VAL** value but does not modify **errno**.

prefresh or pnoutrefresh Subroutine

Purpose

Updates the terminal and curscr (current screen) to reflect changes made to a pad.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
prefresh(Pad, PY, PX, TTY, TTX, TBY, TBX)
WINDOW * Pad;
int  PY,  PX,  TTY;
int  TTX,  TBY,  TBX;
```

```
pnoutrefresh(Pad, PY, PX, TTY, TTX, TBY, TBX)
WINDOW *Pad;
int PY, PX, TTY;
int TTX, TBY, TBX;
```

Description

The **prefresh** and **pnoutrefresh** subroutines are similar to the **wrefresh** (“refresh or wrefresh Subroutine” on page 1728) and **wnoutrefresh** (“doupdate, refresh, wnoutrefresh, or wrefresh Subroutines” on page 255) subroutines. They are different in that pads, instead of windows, are involved, and additional parameters are necessary to indicate what part of the pad and screen are involved.

The *PX* and *PY* parameters specify the upper left corner, in the pad, of the rectangle to be displayed. The *TTX*, *TTY*, *TBX*, and *TBY* parameters specify the edges, on the screen, for the rectangle to be displayed in. The lower right corner of the rectangle to be displayed is calculated from the screen coordinates, since both rectangle and pad must be the same size. Both rectangles must be entirely contained within their respective structures.

The **prefresh** subroutine copies the specified portion of the pad to the physical screen. if you wish to output several pads at once, call **pnoutrefresh** for each pad and then issue one call to **doupdate**. This updates the physical screen once.

Parameters

Item	Description
Pad	Specifies the pad to be refreshed.
PX	(Pad's x-coordinate) Specifies the upper-left column coordinate, in the pad, of the rectangle to be displayed.
PY	(Pad's y-coordinate) Specifies the upper-left row coordinate, in the pad, of the rectangle to be displayed.

Item	Description
TBX	(Terminal's Bottom x-coordinate) Specifies the lower-right column coordinate, on the terminal, for the pad to be displayed in.
TBY	(Terminal's Bottom y-coordinate) Specifies the lower-right row coordinate, on the terminal, for the pad to be displayed in.
TTX	(Terminal's Top x-coordinate) Specifies the upper-left column coordinate, on the terminal, for the pad to be displayed in.
TTY	(Terminal's Top Y coordinate) Specifies the upper-left row coordinate, on the terminal, for the pad to be displayed in.

Examples

1. To update the user-defined my_pad pad from the upper-left corner of the pad on the terminal with the upper-left corner at the coordinates Y=20, X=10 and the lower-right corner at the coordinates Y=30, X=25 enter

```
WINDOW *my_pad;
prefresh(my_pad, 0, 0, 20, 10, 30, 25);
```

2. To update the user-defined my_pad1 and my_pad2 pads and output them both to the terminal in one burst of output, enter:

```
WINDOW *my_pad1; *my_pad2; pnoutrefresh(my_pad1, 0, 0, 20, 10, 30, 25);
pnoutrefresh(my_pad2, 0, 0, 0, 0, 10, 5);
doupdate();
```

printf, fprintf, sprintf, snprintf, wsprintf, vprintf, vfprintf, vsprintf, vwsprintf, or vdprintf Subroutine

Purpose

Prints formatted output.

Library

Standard C Library (**libc.a**) or the Standard C Library with 128-Bit long doubles (**libc128.a**)

Syntax

```
#include <stdio.h>

int printf (Format, [Value, ...])
const char *Format;

int fprintf (Stream, Format, [Value, ...])
FILE *Stream;
const char *Format;

int sprintf (String, Format, [Value, ...])
char *String;
const char *Format;

int snprintf (String, Number, Format, [Value, . . .])
char *String;
int Number;
const char *Format;
```

```
#include <stdarg.h>

int vprintf (Format, Value)
const char *Format;
va_list Value;

int vfprintf (Stream, Format, Value)
FILE *Stream;
const char *Format;
va_list Value;

int vsprintf (String, Format, Value)
char *String;
const char *Format;
va_list Value;

int vdprintf (fildes, Format, Value);
int fildes;
const char *Format;
va_list Value;
```

```
#include <wchar.h>

int vwsprintf (String, Format, Value)
wchar_t *String;
const char *Format;
va_list Value;

int wsprintf (String, Format, [Value, ...])
wchar_t *String;
const char *Format;
```

Description

The **printf** subroutine converts, formats, and writes the *Value* parameter values, under control of the *Format* parameter, to the standard output stream. The **printf** subroutine provides conversion types to handle code points and **wchar_t** wide character codes.

The **fprintf** subroutine converts, formats, and writes the *Value* parameter values, under control of the *Format* parameter, to the output stream specified by the *Stream* parameter. This subroutine provides conversion types to handle code points and **wchar_t** wide character codes.

The **sprintf** subroutine converts, formats, and stores the *Value* parameter values, under control of the *Format* parameter, into consecutive bytes, starting at the address specified by the *String* parameter. The **sprintf** subroutine places a null character (\0) at the end. You must ensure that enough storage space is available to contain the formatted string. This subroutine provides conversion types to handle code points and **wchar_t** wide character codes.

The **snprintf** subroutine converts, formats, and stores the *Value* parameter values, under control of the *Format* parameter, into consecutive bytes, starting at the address specified by the *String* parameter. The **snprintf** subroutine places a null character (\0) at the end. You must ensure that enough storage space is available to contain the formatted string. This subroutine provides conversion types to handle code points and **wchar_t** wide character codes. The **snprintf** subroutine is identical to the **sprintf** subroutine with the addition of the *Number* parameter, which states the size of the buffer referred to by the *String* parameter.

The **wsprintf** subroutine converts, formats, and stores the *Value* parameter values, under control of the *Format* parameter, into consecutive **wchar_t** characters starting at the address specified by the *String* parameter. The **wsprintf** subroutine places a null character (\0) at the end. The calling process should ensure that enough storage space is available to contain the formatted string. The field width unit is specified as the number of **wchar_t** characters. The **wsprintf** subroutine is the same as the **printf** subroutine, except that the *String* parameter for the **wsprintf** subroutine uses a string of **wchar_t** wide-character codes.

All of the above subroutines work by calling the **_doprnt** subroutine, using variable-length argument facilities of the **varargs** macros.

The **vdprintf**, **vprintf**, **vfprintf**, **vsprintf**, and **vwsprintf** subroutines format and write **varargs** macros parameter lists. These subroutines are the same as the **drpintf**, **printf**, **fprintf**, **sprintf**, **snprintf**, and **wsprintf** subroutines, respectively, except that they are not called with a variable number of parameters. Instead, they are called with a parameter-list pointer as defined by the **varargs** macros.

Note: Starting with the IBM AIX 6 with Technology Level 7 and the IBM AIX 7 with Technology Level 1, the precision of the floating-point conversion routines, printf and scanf family of functions has been increased from 17 digits to 37 digits for double and long double values.

Parameters

Number
Specifies the number of bytes in a string to be copied or transformed.

Value
Specifies 0 or more arguments that map directly to the objects in the *Format* parameter.

Stream
Specifies the output stream.

String
Specifies the starting address.

Format
A character string that contains two types of objects:

- Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes 0 or more items to be retrieved from the *Value* parameter list. In the case of the **vprintf**, **vfprintf**, **vsprintf**, and **vwsprintf** subroutines, each conversion specification causes 0 or more items to be retrieved from the **varargs** macros parameter lists.

  If the *Value* parameter list does not contain enough items for the *Format* parameter, the results are unpredictable. If more parameters remain after the entire *Format* parameter has been processed, the subroutine ignores them.

  Each conversion specification in the *Format* parameter has the following elements:

- A % (percent sign).
- 0 or more options, which modify the meaning of the conversion specification. The option characters and their meanings are:

  '
  Formats the integer portions resulting from **i**, **d**, **u**, **f**, **g** and **G** decimal conversions with **thousands_sep** grouping characters. For other conversions the behavior is undefined. This option uses the nonmonetary grouping character.

  -
  Left-justifies the result of the conversion within the field.

  +
  Begins the result of a signed conversion with a + (plus sign) or - (minus sign).

  space character
  Prefixes a space character to the result if the first character of a signed conversion is not a sign. If both the space-character and **+** option characters appear, the space-character option is ignored.

  #
  Converts the value to an alternate form. For **c**, **d**, **s**, and **u** conversions, the option has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a 0. For **x** and **X** conversions, a nonzero result has a 0x or 0X prefix. For **e**, **E**, **f**, **g**, and **G** conversions, the result always contains a decimal point, even if no digits follow it. For **g** and **G** conversions, trailing 0's are not removed from the result.

  0
  Pads to the field width with leading 0's (following any indication of sign or base) for **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions; the field is not space-padded. If the **0** and **-** options both appear, the **0** option is ignored. For **d**, **i**, **o u**, **x**, and **X** conversions, if a precision is specified, the **0** option is also ignored. If the **0** and **'** options both appear, grouping characters are inserted before the field is padded. For other conversions, the results are unreliable.

  B
  Specifies a no-op character.

  N
  Specifies a no-op character.

  J
  Specifies a no-op character.
- An optional decimal digit string that specifies the minimum field width. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the **-** (left-justify) option is specified, the field is padded on the right.
- An optional precision. The precision is a **.** (dot) followed by a decimal digit string. If no precision is specified, the default value is 0. The precision specifies the following limits:

  - Minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions.
  - Number of digits to appear after the decimal point for the **e**, **E**, and **f** conversions.
  - Maximum number of significant digits for **g** and **G** conversions.
  - Maximum number of bytes to be printed from a string in **s** and **S** conversions.
  - Maximum number of bytes, converted from the **wchar_t** array, to be printed from the **S** conversions. Only complete characters are printed.
- An optional **l** (lowercase *L*), **ll** (lowercase *LL*), **h**, or **L** specifier indicates one of the following:
  - An optional **h** specifying that a subsequent **d**, **i**, **u**, **o**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** *Value* parameter (the parameter will have been promoted according to the integral promotions, and its value will be converted to a **short int** or **unsigned short int** before printing).
  - An optional **h** specifying that a subsequent **n** conversion specifier applies to a pointer to a **short int** parameter.
  - An optional **l** (lowercase *L*) specifying that a subsequent **d**, **i**, **u**, **o**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** parameter .
  - An optional **l** (lowercase *L*) specifying that a subsequent **n** conversion specifier applies to a pointer to a **long int** parameter.
  - An optional **ll** (lowercase *LL*) specifying that a subsequent **d**, **i**, **u**, **o**, **x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** parameter.
  - An optional **ll** (lowercase *LL*) specifying that a subsequent **n** conversion specifier applies to a pointer to a **long long int** parameter.
  - An optional **L** specifying that a following **e**, **E**, **f**, **g**, or **G** conversion specifier applies to a **long double** parameter. If linked with **libc.a**, **long double** is the same as double (64bits). If linked with **libc128.a** and **libc.a**, **long double** is 128 bits.
- An optional **H**, **D**, or **DD** specifier indicates one of the following conversions:
  - An optional **H** specifying that a following **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a **_Decimal32** parameter.
  - An optional **D** specifying that a following **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a **_Decimal64** parameter.
  - An optional **DD** specifying that a following **e**, **E**, **f**, **F**, **g**, or **G** conversion specifier applies to a **_Decimal128** parameter.
- An optional `vl`, `lv`, `vh`, `hv` or `v` specifier indicates one of the following vector data type conversions:
  - An optional `v` specifying that a following `e`, `E`, `f`, `g`, `G`, `a`, or `A` conversion specifier applies to a `vector float` parameter. It consumes one argument and interprets the data as a series of four 4-byte floating point components.
  - An optional `v` specifying that a following `c`, `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to a `vector signed char`, `vector unsigned char`, or `vector bool char` parameter. It consumes one argument and interprets the data as a series of sixteen 1-byte components.
  - An optional `vl` or `lv` specifying that a following `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to a `vector signed int`, `vector unsigned int`, or `vector bool` parameter. It consumes one argument and interprets the data as a series of four 4-byte integer components.
  - An optional `vh` or `hv` specifying that a following `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to a `vector signed short` or `vector unsigned short` parameter. It consumes one argument and interprets the data as a series of eight 2-byte integer components.
  - For any of the preceding specifiers, an optional separator character can be specified immediately preceding the vector size specifier. If no separator is specified, the default separator is a space unless the conversion is `c`, in which case the default separator is null. The set of supported optional separators are `,` (comma), `;` (semicolon), `:` (colon), and `_` (underscore).
- The following characters indicate the type of conversion to be applied:

%
Performs no conversion. Prints (**%**).

d or i
Accepts a *Value* parameter specifying an integer and converts it to signed decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field-width value to be padded with leading 0's.

u
Accepts a *Value* parameter specifying an unsigned integer and converts it to unsigned decimal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field-width value to be padded with leading 0's.

o
Accepts a *Value* parameter specifying an unsigned integer and converts it to unsigned octal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field-width with a 0 as a leading character causes the field width value to be padded with leading 0's. An octal value for field width is not implied.

x or X
Accepts a *Value* parameter specifying an unsigned integer and converts it to unsigned hexadecimal notation. The letters **abcdef** are used for the **x** conversion and the letters **ABCDEF** are used for the **X** conversion. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading 0's. The default precision is 1. The result of converting a value of 0 with a precision of 0 is a null string. Specifying a field width with a 0 as a leading character causes the field-width value to be padded with leading 0's.

f
Accepts a *Value* parameter specifying a double and converts it to decimal notation in the format [-]*ddd***.***ddd*. The number of digits after the decimal point is equal to the precision specification. If no precision is specified, six digits are output. If the precision is 0, no decimal point appears.

e or E
Accepts a *Value* parameter specifying a double and converts it to the exponential form [-]*d.ddd***e**+/-*dd*. One digit exists before the decimal point, and the number of digits after the decimal point is equal to the precision specification. The precision specification can be in the range of 0-17 digits. If no precision is specified, six digits are output. If the precision is 0, no decimal point appears. The **E** conversion character produces a number with **E** instead of **e** before the exponent. The exponent always contains at least two digits.

g or G
Accepts a *Value* parameter specifying a double and converts it in the style of the **e**, **E**, or **f** conversion characters, with the precision specifying the number of significant digits. Trailing 0's are removed from the result. A decimal point appears only if it is followed by a digit. The style used depends on the value converted. Style **e** (**E**, if **G** is the flag used) results only if the exponent resulting from the conversion is less than -4, or if it is greater or equal to the precision. If an explicit precision is 0, it is taken as 1.

c
Accepts and prints a *Value* parameter specifying an integer converted to an **unsigned char** data type.

C
Accepts and prints a *Value* parameter specifying a **wchar_t** wide character code. The **wchar_t** wide character code specified by the *Value* parameter is converted to an array of bytes

representing a character and that character is written; the *Value* parameter is written without conversion when using the **wsprintf** subroutine.

s

Accepts a *Value* parameter as a string (character pointer), and characters from the string are printed until a null character (\0) is encountered or the number of bytes indicated by the precision is reached. If no precision is specified, all bytes up to the first null character are printed. If the string pointer specified by the *Value* parameter has a null value, the results are unreliable.

S

Accepts a corresponding *Value* parameter as a pointer to a **wchar_t** string. Characters from the string are printed (without conversion) until a null character (\0) is encountered or the number of wide characters indicated by the precision is reached. If no precision is specified, all characters up to the first null character are printed. If the string pointer specified by the *Value* parameter has a value of null, the results are unreliable.

p

Accepts a pointer to void. The value of the pointer is converted to a sequence of printable characters, the same as an unsigned hexadecimal (x).

n

Accepts a pointer to an integer into which is written the number of characters (wide-character codes in the case of the **wsprintf** subroutine) written to the output stream by this call. No argument is converted.

A field width or precision can be indicated by an * (asterisk) instead of a digit string. In this case, an integer *Value* parameter supplies the field width or precision. The *Value* parameter converted for output is not retrieved until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted.

If the result of a conversion is wider than the field width, the field is expanded to contain the converted result and no truncation occurs. However, a small field width or precision can cause truncation on the right.

The **printf**, **fprintf**, **sprintf**, **snprintf**, **wsprintf**, **vprintf**, **vfprintf**, **vsprintf**, or **vwsprintf** subroutine allows the insertion of a language-dependent radix character in the output string. The radix character is defined by language-specific data in the **LC_NUMERIC** category of the program's locale. In the C locale, or in a locale where the radix character is not defined, the radix character defaults to a . (dot).

After any of these subroutines runs successfully, and before the next successful completion of a call to the **fclose** or **fflush** subroutine on the same stream or to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

The **e**, **E**, **f**, **g**, and **G** conversion specifiers represent the special floating-point values as follows:

Item	Description
Quiet NaN	+NaNQ or -NaNQ
Signaling NaN	+NaNS or -NaNS
+/-INF	+INF or -INF
+/-0	+0 or -0

The representation of the + (plus sign) depends on whether the **+** or space-character formatting option is specified.

These subroutines can handle a format string that enables the system to process elements of the parameter list in variable order. In such a case, the normal conversion character % (percent sign) is replaced by **%***digit***$**, where *digit* is a decimal number in the range from 1 to the **NL_ARGMAX** value. Conversion is then applied to the specified argument, rather than to the next unused argument. This feature provides for the definition of format strings in an order appropriate to specific languages. When

variable ordering is used the * (asterisk) specification for field width in precision is replaced by **%***digit***$**. If you use the variable-ordering feature, you must specify it for all conversions.

The following criteria apply:

- The format passed to the NLS extensions can contain either the format of the conversion or the explicit or implicit argument number. However, these forms cannot be mixed within a single format string, except for %% (double percent sign).
- The *n* value must have no leading zeros.
- If **%***n***$** is used, **%**1**$** to **%***n* - 1**$** inclusive must be used.
- The *n* in **%***n***$** is in the range from 1 to the **NL_ARGMAX** value, inclusive. See the **limits.h** file for more information about the **NL_ARGMAX** value.
- Numbered arguments in the argument list can be referenced as many times as required.
- The * (asterisk) specification for field width or precision is not permitted with the variable order **%***n***$** format; instead, the ***m***$** format is used.

Return Values

Upon successful completion, the **printf**, **fprintf**, **vprintf**, and **vfprintf** subroutines return the number of bytes transmitted (not including the null character [\0] in the case of the **sprintf**, and **vsprintf** subroutines). If an error was encountered, a negative value is output.

Upon successful completion, the **snprintf** subroutine returns the number of bytes written to the *String* parameter (excluding the terminating null byte). If output characters are discarded because the output exceeded the *Number* parameter in length, then the **snprintf** subroutine returns the number of bytes that would have been written to the *String* parameter if the *Number* parameter had been large enough (excluding the terminating null byte).

Upon successful completion, the **wsprintf** and **vwsprintf** subroutines return the number of wide characters transmitted (not including the wide character null character [\0]). If an error was encountered, a negative value is output.

Error Codes

The **printf**, **fprintf**, **sprintf**, **snprintf**, or **wsprintf** subroutine is unsuccessful if the file specified by the *Stream* parameter is unbuffered or the buffer needs to be flushed and one or more of the following are true:

Item	Description
EAGAIN	The **O_NONBLOCK** or O_NDELAY flag is set for the file descriptor underlying the file specified by the *Stream* or *String* parameter and the process would be delayed in the write operation.
EBADF	The file descriptor underlying the file specified by the *Stream* or *String* parameter is not a valid file descriptor open for writing.
EFBIG	An attempt was made to write to a file that exceeds the file size limit of this process or the maximum file size. For more information, refer to the **ulimit** subroutine.
EINTR	The write operation terminated due to receipt of a signal, and either no data was transferred or a partial transfer was not reported. **Note:** Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding **sa_restart**.

Item	Description
EIO	The process is a member of a background process group attempting to perform a write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal, and the process group of the process has no parent process.
ENOSPC	No free space remains on the device that contains the file.
EOVERFLOW	In the UNIX03 mode, the **snprintf** or **vsnprintf** subroutine is unsuccessful if the value of Number parameter is greater than the value of **INT_MAX**. **Note:** The UNIX03 behavior is enabled, if the value of the **XPG_SUS_ENV** environment variable is set to ON.
EPIPE	An attempt was made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal is sent to the process.

The **printf**, **fprintf**, **sprintf**, **snprintf**, or **wsprintf** subroutine may be unsuccessful if one or more of the following are true:

Item	Description
EILSEQ	An invalid character sequence was detected.
EINVAL	The *Format* parameter received insufficient arguments.
ENOMEM	Insufficient storage space is available.
ENXIO	A request was made of a nonexistent device, or the request was outside the capabilities of the device.

Examples

The following example demonstrates how the **vfprintf** subroutine can be used to write an error routine:

```
#include <stdio.h>
#include <stdarg.h>
/* The error routine should be called with the
     syntax:       */
/* error(routine_name, Format
     [, value, . . . ]); */
/*VARARGS0*/
void error(char *fmt, . . .);
/* **  Note that the function name and
     Format arguments cannot be **
     separately declared because of the **
     definition of varargs.  */  {
   va_list args;

   va_start(args, fmt);
   /*
   ** Display the name of the function
      that called the error routine   */
   fprintf(stderr, "ERROR in %s: ",
      va_arg(args, char *));   /*
   ** Display the remainder of the message
   */
   fmt = va_arg(args, char *);
   vfprintf(fmt, args);
   va_end(args);
    abort();  }
```

printw, wprintw, mvprintw, or mvwprintw Subroutine

Purpose

Performs a **printf** command on a window using the specified format control string.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

printw(*Format*, [*Argument ...*])
char ***Format*, ***Argument*;

wprintw(*Window*, *Format*, [*Argument ...*])
WINDOW ***Window*;**
char ***Format*, ***Argument*;**

mvprintw(*Line*, *Column*, *Format*, [*Argument ...*])
int *Line, Column*;
char ***Format*, ***Argument*;**

```
mvwprintw(Window, Line, Column, Format, [Argument ...])

WINDOW *Window;
int Line, Column;
char *Format, *Argument;
```

Description

The **printw**, **wprintw**, **mvprintw**, and **mvwprintw** subroutines perform output on a window by using the specified format control string. However, the **waddch** (“addch, mvaddch, mvwaddch, or waddch Subroutine ” on page 41) subroutine is used to output characters in a given window instead of invoking the **printf** subroutine. The **mvprintw** and **mvwprintw** subroutines move the logical cursor before performing the output.

Use the **printw** and **mvprintw** subroutines on the stdscr and the **wprintw** and **mvwprintw** subroutines on user-defined windows.

Note: The maximum length of the format control string after expansion is 512 bytes.

Parameters

Item	Description
Argument	Specifies the item to print. See the **printf** subroutine for more details.
Column	Specifies the horizontal position to move the cursor to before printing.
Format	Specifies the format for printing the *Argument* parameter. See the **printf** subroutine.
Line	Specifies the vertical position to move the cursor to before printing.
Window	Specifies the window to print into.

Examples

1. To print the user-defined integer variables x and y as decimal integers in the stdscr, enter:

   ```
   int x, y;
   printw("%d%d", x, y);
   ```

2. To print the user-defined integer variables x and y as decimal integers in the user-defined window `my_window`, enter:

```
int x, y;
WINDOW *my_window;
wprintw(my_window, "%d%d", x, y);
```

3. To move the logical cursor to the coordinates y = 5, x = 10 before printing the user-defined integer variables x and y as decimal integers in the stdscr, enter:

```
int x, y;
mvprintw(5, 10, "%d%d", x, y);
```

4. To move the logical cursor to the coordinates y = 5, x = 10 before printing the user-defined integer variables x and y as decimal integers in the user-defined window `my_window`, enter:

```
int x, y;
WINDOW *my_window;
mvwprintw(my_window, 5, 10, "%d%d", x, y);
```

priv_clrall Subroutine

Purpose

Removes all of the privilege bits from the privilege set.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

void priv_clrall(privg_t pv)
```

Description

The **priv_clrall** subroutine removes all of the privilege bits in the privilege set specified by the *pv* parameter.

Parameters

Item	Description
pv	Specifies the privilege set.

Return Values

The **priv_clrall** subroutine returns no values.

Errors

No **errno** value is set.

priv_comb Subroutine

Purpose

Computes the union of privilege sets.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

void priv_comb (privg_t pv1, privg_t pv2, privg_t pv3)
```

Description

The **priv_comb** subroutine computes the union of the privileges specified in the *pv1* and *pv2* parameters and stores the result in the *pv3* parameter.

Parameters

Item	Description
pv1	Specifies the privilege set.
pv2	Specifies the privilege set.
pv3	Specifies the privilege set to store.

Return Values

The **priv_comb** subroutine returns no values.

Errors

No **errno** value is set.

priv_copy Subroutine

Purpose

Copies privileges.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

void priv_copy(privg_t pv1, privg_t pv2)
```

Description

The **priv_copy** subroutine copies all of the privileges specified in the *pv1* privilege set to the *pv2* privilege set, and replaces all of the privileges in the *pv2* privilege set.

Parameters

Item	Description
pv1	Specifies the privilege set to copy from.
pv2	Specifies the privilege set to copy to.

Return Values

The **priv_copy** subroutine returns no values.

Errors

No **errno** value is set.

priv_isnull Subroutine

Purpose

Determines if a privilege set is empty.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

int priv_isnull(privg_t pv)
```

Description

The **priv_isnull** subroutine determines whether the privilege set specified by the *pv* parameter is empty. If the *pv* is empty, it returns a value of 1; otherwise, it returns a value of zero.

Parameters

Item	Description
pv	Specifies the privilege set.

Return Values

The **priv_isnull** subroutine returns one of the following values:

Item	Description
0	The value of the *pv* parameter is not empty.
1	The value of the *pv* parameter is empty.

Errors

No **errno** value is set.

priv_lower Subroutine

Purpose

Removes the privilege from the effective privilege set of the calling process.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

int priv_lower (int priv1, ...)
```

Description

The **priv_lower** subroutine removes each of the privileges in the comma separated privilege list from the effective privilege set of the calling process. The argument list beginning with the *priv1* is of the variable length and must be terminated with a negative value. The numeric values of the privileges are defined in the header file <**sys/priv.h**>. The maximum privilege set, limiting privilege set, and other privileges in the effective privilege set are not affected.

The **priv_lower**, **priv_remove**, and **priv_raise** subroutines all call the **setppriv** subroutine. Thus the calling process of these subroutine is subject to all of the restrictions and privileges imposed by the use of the **setppriv** subroutine.

Parameters

Item	Description
priv1	The privilege identified by its number defined in the <**sys/priv.h**> file.

Return Values

The **priv_lower** subroutine returns one of the following values:

Item	Description
0	The subroutine completes successfully.
1	An error has occurred.

Errors

No **errno** value is set.

priv_mask Subroutine

Purpose

Stores the intersection of two privilege sets into a new privilege set.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

void priv_mask(privg_t pv1, privg_t pv2, privg_t pv3)
```

Description

The **priv_mask** subroutine computes the intersection of the privilege set specified by the *pv1* and *pv2* parameters, and stores the result into the *pv3* parameter.

Parameters

Item	Description
pv1	Specifies the privilege set.
pv2	Specifies the privilege set.
pv3	Specifies the place to store the intersection of the *pv1* and *pv2* parameters.

Return Values

The **priv_mask** subroutine returns no values.

Errors

No **errno** value is set.

priv_raise Subroutine

Purpose

Adds the privilege to the effective privilege set of the calling process.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

int priv_raise(int priv1, ...)
```

Description

The **priv_raise** adds each of the privileges in the comma separated privilege list to the effective privilege set of the calling process. The argument list beginning with the *priv1* parameter is of the variable length and must be terminated with a negative value. The numeric values of the privileges are defined in the header file <**sys/priv.h**>. To set a privilege in the effective privilege set, the calling process must have the corresponding privilege enabled in its maximum and limiting privilege sets. The **priv_raise** subroutine does not affect the maximum privilege set, limiting privilege set, or other privileges in the effective privilege set.

The **priv_lower**, **priv_remove**, and **priv_raise** subroutines all call the **setppriv** subroutine. Thus the calling process of these subroutine is subject to all of the restrictions and privileges imposed by the use of the **setppriv** subroutine.

Parameters

Item	Description
priv1	The privilege identified by its number defined in the <**sys/priv.h**> file.

Return Values

The **priv_raise** subroutine returns one of the following values:

Item	Description
0	The subroutine completes successfully.
1	An error has occurred.

Errors

No **errno** value is set.

priv_rem Subroutine

Purpose

Removes a subset of a privilege set and copies the privileges to another privilege set.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

void priv_rem(privg_t pv1, privg_t pv2, privg_t pv3)
```

Description

When the privileges in the *pv2* parameter are a subset of the privileges in the *pv1* parameter, the **priv_rem** subroutine removes the privileges in the *pv2* parameter and stores them into the *pv3* parameter.

Parameters

Item	Description
pv1	Specifies the privilege set that contains privileges of the *pv2* parameter.
pv2	Specifies the privilege set that is a subset of the privileges of the *pv1* parameter.
pv3	Specifies the privilege set to store the privileges of the *pv3* parameter.

Return Values

The **priv_rem** subroutine returns no values.

Errors

No **errno** value is set.

priv_remove Subroutine

Purpose

Removes the privilege of the calling process.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

int priv_remove(int priv1, ...)
```

Description

The **priv_remove** subroutine removes each of the privileges in the comma separated privilege list from the effective and maximum privilege sets of the calling process. The argument list beginning with the *priv1* is of the variable length and must be terminated with a negative value. The numeric values of the privileges are defined in the header file <**sys/priv.h**>. This subroutine does not affect the limiting privilege set, or other privileges in the effective and maximum privilege sets.

The **priv_lower**, **priv_remove**, and **priv_raise** subroutines all call the **setppriv** subroutine. Thus the calling process of these subroutine is subject to all of the restrictions and privileges imposed by the use of the **setppriv** subroutine.

Parameters

Item	Description
priv1	The privilege identified by its number defined in the <**sys/priv.h**> file.

Return Values

The **priv_remove** subroutine returns one of the following values:

Item	Description
0	The subroutine completes successfully.
1	An error has occurred.

Errors

No **errno** value is set.

priv_setall Subroutine

Purpose

Sets all privileges in the privilege set.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

void priv_setall(privg_t pv)
```

Description

The **priv_setall** subroutine sets all of the privileges in the privilege set specified by the *pv* parameter.

Parameters

Item	Description
pv	Specifies the privilege set.

Return Values

The **priv_setall** subroutine returns no values.

Errors

No **errno** value is set.

priv_subset Subroutine

Purpose

Determines whether the privileges are subsets.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

int priv_subset(privg_t pv1, privg_t pv2)
```

Description

The **priv_subset** subroutine determines whether the privileges specified by the *pv1* parameter are subsets of the privileges specified by the *pv2* parameter.

Parameters

Item	Description
pv1	The privilege set that might be the subsets of the *pv2* parameter.
pv2	The privilege set whose subsets might be the *pv1* parameter.

Return Values

The **priv_subset** subroutine returns one of the following values:

Item	Description
0	The *pv1* parameter is not subset of the *pv2* parameter.
1	The *pv1* parameter is subset of the *pv2* parameter.

Errors

No **errno** value is set.

privbit_clr Subroutine

Purpose

Removes a privilege from a privilege set.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

void privbit_clr(privg_t pv, int priv)
```

Description

The **privbit_clr** subroutine removes the privilege specified by the *priv* parameter from the privilege set specified by the *pv* parameter.

Parameters

Item	Description
pv	Specifies the privilege set that the privilege is removed from.
priv	Specifies the privilege to be removed.

Return Values

The **privbit_clr** subroutine returns no values.

Errors

No **errno** value is set.

privbit_set Subroutine

Purpose

Adds a privilege to a privilege set.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

void privbit_set(privg_t pv, int priv)
```

Description

The **privbit_set** subroutine adds the privilege specified by the *priv* parameter into the privilege set specified by the *pv* parameter.

Parameters

Item	Description
priv	Specifies the privilege to add.
pv	Specifies the target privilege set.

Return Values

The **privbit_set** subroutine returns no value.

Errors

No **errno** value is set.

privbit_test Subroutine

Purpose

Determines if a privilege belongs to a privilege set.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpriv.h>
#include <sys/priv.h>

int privbit_test(privg_t pv, int priv)
```

Description

The **privbit_test** subroutine determines whether the privilege specified by the *priv* parameter is contained within the privilege set specified by the *pv* parameter.

Parameters

Item	Description
pv	Specifies the privilege set.

Item	Description
priv	Specifies the privilege.

Return Values

The **privbit_test** subroutine returns one of the following values:

Item	Description
0	The value of the *priv* parameter is not contained within the value of the *pv* parameter.
1	The value of the *priv* parameter is contained within the value of the *pv* parameter.

Errors

No **errno** value is set.

proc_getattr Subroutine

Purpose

Retrieves selected attributes of a process.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/proc.h>

int proc_getattr (pid,attr,size)
pid_t pid;
procattr_t* attr;
size64_t size;
```

Description

The **proc_getattr** subroutines allows you to retrieve the current state of certain process attributes. The information is returned in the **procattr_t** structure defined in the **<sys/proc.h>** header file.

```
typedef struct {
                uchar core_naming; /* Unique core file names */
                uchar core_mmap;   /* Dump nonanonymous mmap regions to core file */
                uchar core_shm;    /* Dump shared memory to core file */
                uchar aixthread_hrt; /* High resolution timer for thread */
               }procattr_t;
```

To retrieve information about the calling process, a -1 can be passed as the first argument, *pid*.

Process A can retrieve process attribute information about Process B if one or more of the following items are true:

- Process A and Process B have the same real or effective user ID.
- Process A was executed by the root user.
- Process A has the **PV_DAC_R** privilege.

Parameters

Item	Description
pid	Specified the process identifier of the process for which the information is to be retrieved.
attr	Specifies apointer to the user structure that holds the information retrieved from the process kernel structure.
size	The **sizeof procattr_t** structure is stored in the *size* parameter when calling the API.

Return Values

Item	Description
0	**proc_getattr** was successful.
-1	**proc_getattr** was unsuccessful. Global variable **errno** is set to indicate the error.

Error Codes

Item	Description
EINVAL	The *size* argument does not match the size of the **procattr_t** in the kernel.
EFAULT	The *attr* value that was passed to the buffer is invalid.
ESRCH	The process identifier could not be located.
EPERM	The privileges are insufficient to read attributes from the target **proc** structure.

Example

```
#include <stdio.h>
#include <sys/proc.h>

dispprocflags.c:
#define P(_x_)  (((_x_) == PA_ENABLE) ? "ENABLE" : \
                ((_x_) == PA_DISABLE ? "DISABLE" : \
                ((_x_) == PA_IGNORE) ? "IGNORE" : "JUNK")))
int main(int argc, char *argv[])
{
        int rc;
        procattr_t attr;
        pid_t pid;
        if (argc &lt; ) {
                printf("Syntax: %s <pid>\n", argv[0]);
                exit(-1);
        }
        pid = atoi(argv[1]);
        bzero(&attr, sizeof(procattr_t));
        rc = proc_getattr(pid, &attr, sizeof(procattr_t));
        if (rc) {
                    printf("proc_getattr failed, errno %d\n", errno);
                    exit(-1);
                }
        printf("core_naming %s\n", P(attr.core_naming));
        printf("core_mmap %s\n", P(attr.core_mmap));
        printf("core_shm %s\n", P(attr.core_shm));
        printf("aixthread_hrt %s\n", P(attr.aixthread_hrt));
       }
crash64.c:
#include <stdio.h>
  int main()
       {
        int *p = (int *)0x100;
        pid_t pid = getpid();
        printf("My pid is %d\n", getpid());
        getchar();
        *p = 0x10;
         printf("Done\n");
```

```
        }
# ./crash64 &  [2]
5570812
# My pid is 5570812
# ./dispcoreflags 5570812
PID 5500FC
core_naming ENABLE
core_mmap ENABLE
core_shm ENABLE
aixthread_hrt DISABLE
# fg  ./crash64
Memory fault(coredump)
# ls core*
core.5570812.11054349
```

proc_mobility_base_set Subroutine

Purpose

Sets or unsets attributes used by AIX Live Update to indicate that the current process is a base process.

Library

Standard C library (*libc.a*)

Syntax

#include <sys/mobility.h>

```
int  proc_mobility_base_set (pid , flag),
pid_t pid;
int flag;
```

Description

The **proc_mobility_base_set** subroutine can be used to register the calling process as a base process for a Live Update operation.

Base processes are those that are not saved and migrated during a Live Update operation. The base processes are left behind on the original logical partition (LPAR), rather than being migrated to the surrogate LPAR.

Only a process that is a child of the init process can be registered as a base process. Otherwise, error code EINVAL is returned.

proc_mobility_base_set subroutine can be used to register a base process only while a Live Kernel Update (LKU), is in progress. If there is no LKU in progress, error code EAGAIN is returned

Parameters

Item	Description
pid	Process ID to act upon. The value 0 indicates the current process. If a non-zero value is specified, it must match the PID of the calling process.
flag	**MOBILITY_BASE_PROCESS** flag sets the base attribute. The value 0 is used to unset the base attribute.

Return Values

Item	Description
0	Success

Item	Description
1	Error

Error Codes

Error Code	Description
ENOSYS	No mobility system in place.
ESRCH	No such process.
EINVAL	Input arguments not valid.
EAGAIN	No LKU, is under progress

Example

The following example shows the usages of the **proc_mobility_base_set** subroutine:

```
#include <stdio.h>
#include <sys/mobility.h>
int main(int argc, char *argv[])
{
    int rc = 0;
    pid_t pid = getpid();

    /* Mark this process as a base process */
    rc = proc_mobility_base_set(0, MOBILITY_BASE_PROCESS);

    if (rc) {
        printf("proc_mobility_base_set failed, errno %d\n", errno);
        exit(-1);
    }

    printf("Process %d is now marked as a base process.\n", pid);

}
```

proc_mobility_restartexit_set Subroutine

Purpose

Sets or unsets attributes used by AIX Live Update to indicate that the current process is a `exit on restart` process.

Library

Standard C library (*libc.a*)

Syntax

#include <stdio.h>

#include <sys/mobility.h>

```
int  proc_mobility_restartexit_set (pid, value, flag),
pid_t pid;
int value;
int flag;
```

Description

The **proc_mobility_restartexit_set** subroutine can be used to register the calling process as a `exit on restart` process for a Live Update operation. The `exit on restart` processes are frozen on the original logical partition (LPAR) but theLive Update operation does not checkpoint their state. These processes are recreated on the surrogate LPAR. When they are restarted, they call the **exit()** function and terminate. Applications which do not have specific state information that must be preserved might choose this method. These applications are not required to release resources that are not supported by the mobility operation. If these applications are monitored by a daemon mechanism, the exit may cause a new instance to start on the surrogate LPAR.

Depending on the flags specified, the process can be marked `exit on restart` for a Live Update operation, or for a workload partition mobility operation, or for both.

Parameters

Item	Description
pid	Process ID to act upon. The value 0 indicates the current process. If a non-zero value is specified, it must match the PID of the calling process.
value	**MOBILITY_RESTART_EXIT** flag sets the `exit on restart` attribute. The value 0 is used to unset the `exit on restart` attribute.
flag	The scope for the attribute are: **PROC_MOBILITY_GLOBAL** If the process is `exit on restart` for the Live Update operation **PROC_MOBILITY_WPAR** If the process is `exit on restart` for Workload Partition (WPAR) mobility.

Return Values

Item	Description
0	Success
1	Error

Error Codes

Error Code	Description
ENOSYS	No mobility system in place.
ESRCH	No such process.
EINVAL	Input arguments not valid.

Example

The following example shows the usages of the **proc_mobility_restartexit_set** subroutine:

```
#include <stdio.h>
#include <sys/mobility.h>
int main(int argc, char *argv[])
{
    int rc = 0;
    pid_t pid = getpid();

    /* Mark this process as “exit on restart” for  live update */
    rc = proc_mobility_restartexit_set(0, MOBILITY_RESTART_EXIT, PROC_MOBILITY_GLOBAL);

    if (rc) {
        printf(“proc_mobility_restartexit_set failed, errno %d\n”, errno);
```

```
        exit(-1);
    }

    printf("Process %d is now marked to exit on restart during an AIX live update.\n", pid);

}
```

proc_setattr Subroutine

Purpose

Sets selected attributes of a process.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/proc.h>
int proc_setattr (pid,attr,size)
pid_t pid;
procattr_t* attr;
size64_t size;
```

Description

The **proc_setattr** subroutines allows you to set selected attributes of a process. The list of selected attributes is defined in the**procattr_t** structure defined in the **<sys/proc.h>** header file.

```
typedef struct {
     uchar core_naming; /* Unique core file names */
     uchar core_mmap;   /* Dump nonanonymous mmap regions to core file */
     uchar core_shm;    /* Dump shared memory to core file */
     uchar aixthread_hrt; /* High resolution timer for thread */
   }procattr_t;
```

To set attributes for the calling process, a -1 can be passed as the first argument, **pid**.

Process A can set process attributes for Process B if one or more of the following items are true:

- Process A and Process B have the same real or effective user ID.
- Process A was executed by the root user.
- Process A has **PV_DAC_W** privilege.

Parameters

Item	Description
pid	The identifier of the process whose information is to be retrieved.
attr	A pointer to the user structure that will hold the information retrieved from the process kernel structure.
size	The **sizeof procattr_t** structure is stored in the *size* parameter when calling API.

Return Values

Item	Description
0	**proc_setattr** was successful.
-1	**proc_setattr** was unsuccessful. Global variable **errno** is set to indicate the error.

Error Codes

Item	Description
EINVAL	The *size* argument does not match the size of the **procattr_t** in the kernel.
EFAULT	The *attr* value passed to the buffer is invalid.
ESRCH	Could not locate the process identifier.
EPERM	Insufficient privileges to read attributes from target the **proc** structure.

Example

```
setprocflags.c
#include <stdio.h>
#include <sys/proc.h>
#define P(_x_)  (((_x_) == PA_ENABLE) ? "ENABLE" : \
                ((_x_) == PA_DISABLE ? "DISABLE" : \
                (((_x_) == PA_IGNORE) ? "IGNORE" : "JUNK")))
int main(int argc, char *argv[])
{
        int rc;
        procattr_t attr;
        pid_t pid;
        int naming,mmap,shm = 0;
        if (argc &lt; ) {
                printf("Syntax: %s <pid> <corenaming> <coremmap> <coreshm>\n", argv[0]);
                exit(-1);
        }
        pid = atoi(argv[1]);
        bzero(&attr, sizeof(procattr_t));
        attr.core_naming = atoi(argv[2]);
        attr.core_mmap = atoi(argv[3]);
        attr.core_shm = atoi(argv[4]);
        rc = proc_setattr(pid, &attr, sizeof(procattr_t));
        if (rc)
                {
                  printf("proc_getattr failed, errno %d\n", errno);
                  exit(-1);
                }
        bzero(&attr, sizeof(procattr_t));
        rc = proc_getattr(pid, &attr, sizeof(procattr_t));
        if (rc)
             {
             printf("proc_getattr failed, errno %d\n", errno);
             exit(-1);
             }
        printf("core_naming %s\n", P(attr.core_naming));
        printf("core_mmap %s\n", P(attr.core_mmap));
        printf("core_shm %s\n", P(attr.core_shm));
        printf("aixthread_hrt %s\n", P(attr.aixthread_hrt));
     }
crash64.c
#include <stdio.h>
  int main()
    {
       int *p = (int *)0x100;
       pid_t pid = getpid();
       printf("My pid is %d\n", getpid());
       getchar();
       *p = 0x10;
       printf("Done\n");
  }
# ./crash64 &
[1]     5570566
# My pid is 5570566
PID 5500FC
# ./setcoreflags 5570566 1 1 1
core_naming ENABLE
core_mmap ENABLE
core_shm ENABLE
aixthread_hrt DISABLE
# fg  ./crash64
Memory fault(coredump)
# ls core*
core.5570566.11054349
```

proc_rbac_op Subroutine

Purpose

Sets, unsets, and queries a process' RBAC properties.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/cred.h>
#include <sys/types.h>

int proc_rbac_op (Pid,Cmd, Param)
pid_t Pid
int Cmd
int *Param
```

Description

The **proc_rbac_op** subroutine is used to set, unset, and query a process' Role Based Access Control (RBAC) awareness.

To use the **proc_rbac_op** subroutine, the calling process must have the **ACT_P_SET_PAGRBAC** privilege. If running in a Trusted AIX environment, the calling process must have the appropriate label properties to perform the operation on the target process specified by the *Pid* parameter.

Parameters

Item	Description
Cmd	Specifies the command to run on the target process. The *Cmd* parameter has the following values: **PROC_RBAC_SET** Sets the flag that is specified in the *Param* parameter for the target process. **PROC_RBAC_UNSET** Clears the flag that is specified in the *Param* parameter for the target process. **PROC_RBAC_GET** Returns the status of the process's security flags in regards to the SEC_NOEXEC, SEC_RBACAWARE, and SEC_PRIVCMD.
Pid	Specifies the Pid for the target process. A negative Pid value denotes the current process.
Param	This parameter is dependent on the command that the *Cmd* parameter specifies. **PROC_RBAC_SET** and **PROC_RBAC_UNSET**: Can only be SEC_NOEXEC or SEC_RBACAWARE. Only one flag can be specified for a call. **PROC_RBAC_GET**: Upon return, holds the status of SEC_NOEXEC, SEC_RBACAWARE, and SEC_PRIVCMD.

Return Values

On successful completion, the **proc_rbac_op** subroutine returns the value of zero. If the subroutine fails, it returns a value of 1, and the **errno** will be set.

Error Codes

The **proc_rbac_op** subroutine fails if one of the following values is true:

Item	Description
EINVAL	An invalid *Cmd* value was given or a NULL pointer was given for the *Status* parameter with the **PROC_RBAC_GET** command.
ESRCH	The *pid* value does not correspond to a valid process.
EPERM	The calling process does not have the appropriate RBAC privilege. Or, if the Trusted AIX is enabled, the calling process does not have the appropriate label information.
EFAULT	The copy operation to the *Param* buffer fails.
ENOSYS	The system is not running in the enhanced RBAC mode.

profil Subroutine

Purpose

Starts and stops program address sampling for execution profiling.

Library

Standard C Library (**libc.a**)

Syntax

#include <mon.h>

void profil (*ShortBuffer*, *BufferSize*, *Offset*, *Scale***)** OR **void profil (** *ProfBuffer*, **-1, 0, 0)**

unsigned short ***ShortBuffer; struct prof ***ProfBuffer**; unsigned int** *Buffersize*, *Scale***; unsigned long** *Offset***;**

Description

The **profil** subroutine arranges to record a histogram of periodically sampled values of the calling process program counter. If *BufferSize* is not -1:

- The parameters to the **profil** subroutine are interpreted as shown in the first syntax definition.
- After this call, the program counter (pc) of the process is examined each clock tick if the process is the currently active process. The value of the *Offset* parameter is subtracted from the pc. The result is multiplied by the value of the *Scale* parameter, shifted right 16 bits, and rounded up to the next half-word aligned value. If the resulting number is less than the *BufferSize* value divided by **sizeof(short)**, the corresponding **short** inside the *ShortBuffer* parameter is incremented. If the result of this increment would overflow an unsigned short, it remains USHRT_MAX.
- The least significant 16 bits of the *Scale* parameter are interpreted as an unsigned, fixed-point fraction with a binary point at the left. The most significant 16 bits of the *Scale* parameter are ignored. For example:

Octal	Hex	Meaning
0177777	0xFFFF	Maps approximately each pair of bytes in the instruction space to a unique **short** in the *ShortBuffer* parameter.
077777	0x7FFF	Maps approximately every four bytes to a **short** in the *ShortBuffer* parameter.
02	0x0002	Maps all instructions to the same location, producing a noninterrupting core clock.
01	0x0001	Turns profiling off.
00	0x0000	Turns profiling off.

Note: Mapping each byte of the instruction space to an individual**short** in the *ShortBuffer* parameter is not possible.

- Profiling, using the first syntax definition, is rendered ineffective by giving a value of 0 for the *BufferSize* parameter.

If the value of the *BufferSize* parameter is -1:

- The parameters to the **profil** subroutine are interpreted as shown in the second syntax definition. In this case, the *Offset* and *Scale* parameters are ignored, and the *ProfBuffer* parameter points to an array of **prof** structures. The **prof** structure is defined in the **mon.h** file, and it contains the following members:

```
caddr_t          p_low;
caddr_t          p_high;
HISTCOUNTER     *p_buff;
int              p_bufsize;
uint             p_scale;
```

If the `p_scale` member has the value of -1, a value for it is computed based on `p_low`, `p_high`, and `p_bufsize`; otherwise `p_scale` is interpreted like the scale argument in the first synopsis. The `p_high` members in successive structures must be in ascending sequence. The array of structures is ended with a structure containing a `p_high` member set to 0; all other fields in this last structure are ignored.

The `p_buff` buffer pointers in the array of **prof** structures must point into a single contiguous buffer space.

- Profiling, using the second syntax definition, is turned off by giving a *ProfBuffer* argument such that the `p_high` element of the first structure is equal to 0.

In every case:

- Profiling remains on in both the child process and the parent process after a **fork** subroutine.
- Profiling is turned off when an **exec** subroutine is run.
- A call to the **profil** subroutine is ineffective if profiling has been previously turned on using one syntax definition, and an attempt is made to turn profiling off using the other syntax definition.
- A call to the **profil** subroutine is ineffective if the call is attempting to turn on profiling when profiling is already turned on, or if the call is attempting to turn off profiling when profiling is already turned off.

Parameters

Item	Description
ShortBuffer	Points to an area of memory in the user address space. Its length (in bytes) is given by the *BufferSize* parameter.
BufferSize	Specifies the length (in bytes) of the buffer.

Item	Description
Offset	Specifies the delta of program counter start and buffer; for example, a 0 *Offset* implies that text begins at 0. If the user wants to use the entry point of a routine for the *Offset* parameter, the syntax of the parameter is as follows: ***(long *)***RoutineName*
Scale	Specifies the mapping factor between the program counter and *ShortBuffer.*
ProfBuffer	Points to an array of **prof** structures.

Return Values

The **profil** subroutine always returns a value of 0. Otherwise, the **errno** global variable is set to indicate the error.

Error Codes

The **profil** subroutine is unsuccessful if one or both of the following are true:

Item	Description
EFAULT	The address specified by the *ShortBuffer* or *ProfBuffer* parameters is not valid, or the address specified by a `p_buff` field is not valid. EFAULT can also occur if there are not sufficient resources to pin the profiling buffer in real storage.
EINVAL	The `p_high` fields in the **prof** structure specified by the *ProfBuffer* parameter are not in ascending order.

proj_execve Subroutine

Purpose

Executes an application with the specified project assignment.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

int proj_execve(char * path char *const arg[], char *const env[], projid_t projid, int force);
```

Description

The `proj_execve` system call assigns the requested project ID to the calling process and runs the given program. This subroutine checks whether the caller is allowed to assign the requested project ID to the application, using the available project assignment rules for the caller's user ID, group ID, and application name. If the requested project assignment is not allowed, an error code is returned. However, the user with root authority or advanced accounting administrator capabilities can force the project assignment by setting the *force* parameter to 1.

Parameters

Item	Description
path	Path for the application or program to be run.
arg	List of arguments for the new process.
env	Environment for the new process.
projid	Project ID to be assigned to the new process.
force	Option to override the allowed project list for the application, user, or group.

Return Values

Item	Description
0	Upon success, does not return to the calling process.
-1	The subroutine failed.

Error Codes

Item	Description
EPERM	Permission denied. A user without privileges attempted the call.

projdballoc Subroutine

Purpose

Allocates a project database handle.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

projdballoc(void **handle)
```

Description

The **projdballoc** subroutine allocates a handle to operate on the project database. By default, this *handle* is initialized to operate on the system project database; however, it can be reset with the **projdbfinit** subroutine to reference another project database.

Parameters

Item	Description
handle	Pointer to a void pointer

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	The passed pointer is NULL
ENOMEM	No space left on memory

projdbfinit Subroutine

Purpose

Sets the handle to use a local project database as specified in the dbfile pointer and opens the file with the specified mode.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

projdbfinit(void *handle, char *file, int mode)
```

Description

The **projdbfinit** subroutine sets the specified *handle* to use the specified project definition file. The file is opened in the specified mode. Subsequently, the project database, as represented by the *handle* parameter, will be referenced through file system primitives.

The project database must be initialized before calling this subroutine. The routines **projdballoc** and **projdbfinit** are provided for this purpose. The specified file is opened in the specified mode. File system calls are used to operate on these types of files. The struct **projdb** is filled as follows:

```
projdb.type = PROJ_LOCAL

projdb.fdes = value returned from open() call.
```

If the *file* parameter is NULL, then the system project database is opened.

Parameters

Item	Description
handle	Pointer to handle
file	Indicate the project definition file name

Item	Description
mode	Indicates the mode in which the file is opened

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Passed handle or file is invalid

projdbfree Subroutine

Purpose

Frees an allocated project database handle.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

projdbfree(void *handle)
```

Description

The **projdbfree** subroutine releases the memory allocated to a project database handle. The closure operation is based on the type of project database. If a project database is local, then it is closed using system primitives. The project database must be initialized before calling this subroutine. The routines **projdballoc** and **projdbfinit** are provided for this purpose.

Parameters

Item	Description
handle	Pointer to a void pointer

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Passed pointer is NULL

psdanger Subroutine

Purpose

Defines the amount of free paging space available.

Syntax

```
#include <signal.h>
#include <sys/vminfo.h>
```

blkcnt_t psdanger (*Signal*)
int *Signal*;

Description

The **psdanger** subroutine returns the difference between the current number of free paging-space blocks and the paging-space thresholds of the system.

Parameters

Item	Description
Signal	Defines the signal.

Return Values

If the value of the *Signal* parameter is 0, the return value is the total number of paging-space blocks defined in the system.

If the value of the *Signal* parameter is -1, the return value is the number of free paging-space blocks available in the system.

If the value of the *Signal* parameter is **SIGDANGER**, the return value is the difference between the current number of free paging-space blocks and the paging-space warning threshold. If the number of free paging-space blocks is less than the paging-space warning threshold, the return value is negative.

If the value of the *Signal* parameter is **SIGKILL**, the return value is the difference between the current number of free paging-space blocks and the paging-space kill threshold. If the number of free paging-space blocks is less than the paging-space kill threshold, the return value is negative.

psignal or psiginfo Subroutine or sys_siglist Vector

Purpose

Prints system signal messages to standard error.

Library

Standard C Library (**libc.a**)

Syntax

```
# include <signal.h>

void psignal ( Signal,  String)
int Signal;
const char *String;

void psiginfo ( Info,  String)
const siginfo_t *Info;
const char *String;
```

```
char *sys_siglist[ ];
```

Description

The **psiginfo** and **psignal** subroutine prints a message on **stderr** associated with a signal number. First the *String* parameter is printed, then the name of the signal and a new line character.

The **psiginfo** and **psignal** subroutine does not change the orientation of the standard error stream.

The **psiginfo** and **psignal** subroutine does not change the setting of **errno** if successful.

The **psiginfo** and **psignal** subroutine marks the updates of the last data modification and last file status change timestamps of the file associated with the standard error stream at some time between their successful completion and **exit**, **abort**, or the completion of **fflush** or **fclose** on **stderr**.

To simplify variant formatting of signal names, the **sys_siglist** vector of message strings is provided. The signal number can be used as an index in this table to get the signal name without the new-line character. The **NSIG** defined in the **signal.h** file is the number of messages provided for in the table. It should be checked because new signals may be added to the system before they are added to the table.

Parameters

Item	Description
Info	Points to a valid **siginfo_t**.
Signal	Specifies a signal. The signal number should be among those found in the **signal.h** file.
String	Specifies a string that is printed. Most usefully, the *String* parameter is the name of the program that incurred the signal.

pthdb_attr, pthdb_cond, pthdb_condattr, pthdb_key, pthdb_mutex, pthdb_mutexattr, pthdb_pthread, pthdb_pthread_key, pthdb_rwlock, or pthdb_rwlockattr Subroutine

Purpose

Reports the pthread library objects.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_pthread (pthdb_session_t  session,
                   pthdb_pthread_t * pthreadp,
                   int              cmd)
int pthdb_pthread_key(pthdb_session_t  session,
                      pthread_key_t   * keyp,
                      int              cmd)
int pthdb_attr(pthdb_session_t  session,
               pthdb_attr_t    * attrp,
               int              cmd)
int pthdb_cond (pthdb_session_t  session,
                pthdb_cond_t    * condp,
                int              cmd)
int pthdb_condattr (pthdb_session_t   session,
                    pthdb_condattr_t * condattrp,
                    int               cmd)
int pthdb_key(pthdb_session_t  session,
              pthdb_pthread_t  pthread,
              pthread_key_t   * keyp,
              int              cmd)
int pthdb_mutex (pthdb_session_t  session,
                 pthdb_mutex_t   * mutexp,
                 int              cmd)
int pthdb_mutexattr (pthdb_session_t    session,
                     pthdb_mutexattr_t * mutexattrp,
                     int                cmd)
int pthdb_rwlock (pthdb_session_t  session,
                  pthdb_rwlock_t  * rwlockp,
                  int              cmd)
int pthdb_rwlockattr (pthdb_session_t     session,
                      pthdb_rwlockattr_t * rwlockattrp,
                      int                 cmd)
```

Description

The pthread library maintains internal lists of objects: pthreads, mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, read/write lock attributes, attributes, pthread specific keys, and active keys. The pthread debug library provides access to these lists one element at a time via the functions listed above.

Each one of those functions acquire the next element in the list of objects. For example, the **pthdb_attr** function gets the next attribute on the list of attributes.

A report of **PTHDB_INVALID_***OBJECT* represents the empty list or the end of a list, where *OBJECT* is equal to **PTHREAD**, **ATTR**, **MUTEX**, **MUTEXATTR**, **COND**, **CONDATTR**, **RWLOCK**, **RWLOCKATTR**, **KEY**, or **TID** as appropriate.

Each list is reset to the top of the list when the **pthdb_session_update** function is called, or when the list function reports a **PTHDB_INVALID_*** value. For example, when **pthdb_attr** reports an attribute of **PTHDB_INVALID_ATTR** the list is reset to the beginning such that the next call reports the first attribute in the list, if any.

When **PTHDB_LIST_FIRST** is passed for the *cmd* parameter, the first item in the list is retrieved.

Parameters

Item	Description
session	Session handle.
attrp	Attribute object.
cmd	Reset to the beginning of the list.
condp	Pointer to Condition variable object.
condattrp	Pointer to Condition variable attribute object.
keyp	Pointer to Key object.
mutexattrp	Pointer to Mutex attribute object.
mutexp	Pointer to Mutex object.
pthread	pthread object.
pthreadp	Pointer to pthread object.
rwlockp	Pointer to Read/Write lock object.
rwlockattrp	Pointer to Read/Write lock attribute object.

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_PTHREAD	Invalid pthread handle.
PTHDB_BAD_CMD	Invalid command.
PTHDB_BAD_POINTER	Invalid buffer pointer.
PTHDB_INTERNAL	Error in library.
PTHDB_MEMORY	Not enough memory

pthdb_attr_detachstate,pthdb_attr_addr, pthdb_attr_guardsize,pthdb_attr_inheritsched, pthdb_attr_schedparam,pthdb_attr_schedpolicy,

pthdb_attr_schedpriority,pthdb_attr_scope, pthdb_attr_stackaddr,pthdb_attr_stacksize, or pthdb_attr_suspendstate Subroutine

Purpose

Query the various fields of a pthread attribute and return the results in the specified buffer.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_attr_detachstate (pthdb_session_t      session,
                            pthdb_attr_t         attr,
                            pthdb_detachstate_t * detachstatep);
```

```
int pthdb_attr_addr (pthdb_session_t      session,
                     pthdb_attr_t         attr,
                     pthdb_addr_t * addrp);
```

```
int pthdb_attr_guardsize (pthdb_session_t      session,
                          pthdb_attr_t         attr,
                          pthdb_size_t * guardsizep);
```

```
int pthdb_attr_inheritsched (pthdb_session_t       session,
                             pthdb_attr_t          attr,
                             pthdb_inheritsched_t * inheritschedp);
```

```
int pthdb_attr_schedparam (pthdb_session_t     session,
                           pthdb_attr_t        attr,
                           struct sched_param * schedparamp);
```

```
int pthdb_attr_schedpolicy (pthdb_session_t  session,
                            pthdb_attr_t     attr,
                            pthdb_policy_t  * schedpolicyp)
```

```
int pthdb_attr_schedpriority (pthdb_session_t  session,
                              pthdb_attr_t     attr,
                              int             * schedpriorityp)
```

```
int pthdb_attr_scope (pthdb_session_t  session,
                      pthdb_attr_t     attr,
                      pthdb_scope_t   * scopep)
```

```
int pthdb_attr_stackaddr (pthdb_session_t      session,
                          pthdb_attr_t         attr,
                          pthdb_size_t * stackaddrp);
```

```
int pthdb_attr_stacksize (pthdb_session_t      session,
                          pthdb_attr_t         attr,
                          pthdb_size_t * stacksizep);
```

```
int pthdb_attr_suspendstate (pthdb_session_t       session,
                             pthdb_attr_t          attr,
                             pthdb_suspendstate_t * suspendstatep)
```

Description

Each pthread is created using either the default pthread attribute or a user-specified pthread attribute. These functions query the various fields of a pthread attribute and, if successful, return the result in the buffer specified. In all cases, the values returned reflect the expected fields of a pthread created with the attribute specified.

pthdb_attr_detachstate reports if the created pthread is detachable (**PDS_DETACHED**) or joinable (**PDS_JOINABLE**). **PDS_NOTSUP** is reserved for unexpected results.

pthdb_attr_addr reports the address of the pthread_attr_t.

pthdb_attr_guardsize reports the guard size for the attribute.

pthdb_attr_inheritsched reports whether the created pthread will run with scheduling policy and scheduling parameters from the created pthread (**PIS_INHERIT**), or from the attribute (**PIS_EXPLICIT**). **PIS_NOTSUP** is reserved for unexpected results.

pthdb_attr_schedparam reports the scheduling parameters associated with the pthread attribute. See **pthdb_attr_inheritsched** for additional information.

pthdb_attr_schedpolicy reports whether the scheduling policy associated with the pthread attribute is other (**SP_OTHER**), first in first out (**SP_FIFO**), or round robin (**SP_RR**). **SP_NOTSUP** is reserved for unexpected results.

pthdb_attr_schedpriority reports the scheduling priority associated with the pthread attribute. See **pthdb_attr_inheritsched** for additional information.

pthdb_attr_scope reports whether the created pthread will have process scope (**PS_PROCESS**) or system scope (**PS_SYSTEM**). **PS_NOTSUP** is reserved for unexpected results.

pthdb_attr_stackaddr reports the address of the stack.

pthdb_attr_stacksize reports the size of the stack.

pthdb_attr_suspendstate reports whether the created pthread will be suspended (**PSS_SUSPENDED**) or not (**PSS_UNSUSPENDED**). **PSS_NOTSUP** is reserved for unexpected results.

Parameters

Item	Description
addr	Attributes address.
attr	Attributes handle.
detachstatep	Detach state buffer.
guardsizep	Attribute guard size.
inheritschedp	Inherit scheduling buffer.
schedparamp	Scheduling parameters buffer.
schedpolicyp	Scheduling policy buffer.
schedpriorityp	Scheduling priority buffer.
scopep	Contention scope buffer.
session	Session handle.
stackaddrp	Attributes stack address.
stacksizep	Attributes stack size.
suspendstatep	Suspend state buffer.

Return Values

If successful these functions return **PTHDB_SUCCESS**. Otherwise, and error code is returned.

Error Codes

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_ATTR	Invalid attribute handle.
PTHDB_BAD_POINTER	Invalid buffer pointer.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_NOTSUP	Not supported.
PTHDB_INTERNAL	Internal library error.

pthdb_condattr_pshared, or pthdb_condattr_addr Subroutine

Purpose

Gets the condition variable attribute pshared value.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_condattr_pshared (pthdb_session_t   session,
                            pthdb_condattr_t  condattr,
                            pthdb_pshared_t  * psharedp)

int pthdb_condattr_addr (pthdb_session_t   session,
                         pthdb_condattr_t  condattr,
                         pthdb_addr_t  * addrp)
```

Description

The **pthdb_condattr_pshared** function is used to get the condition variable attribute process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

The **pthdb_condattr_addr** function reports the address of the pthread_condattr_t.

Parameters

Item	Description
addrp	Pointer to the address of the pthread_condattr_t.
condattr	Condition variable attribute handle
psharedp	Pointer to the pshared value.
session	Session handle.

Return Values

If successful this function returns **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_CONDATTR	Invalid condition variable attribute handle.
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INTERNAL	Error in library.
PTHDB_POINTER	Invalid pointer

pthdb_cond_addr, pthdb_cond_mutex or pthdb_cond_pshared Subroutine

Purpose

Gets the condition variable's mutex handle and pshared value.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int  pthdb_cond_addr (pthdb_session_t  session,
                      pthdb_cond_t     cond,
                      pthdb_addr_t * addrp)

int  pthdb_cond_mutex (pthdb_session_t  session,
                       pthdb_cond_t     cond,
                       pthdb_mutex_t * mutexp)

int  pthdb_cond_pshared (pthdb_session_t  session,
                         pthdb_cond_t     cond,
                         pthdb_pshared_t * psharedp)
```

Description

The **pthdb_cond_addr** function reports the address of the pthdb_cond_t.

The **pthdb_cond_mutex** function is used to get the mutex handle associated with the particular condition variable, if the mutex does not exist then PTHDB_INVALID_MUTEX is returned from the mutex.

The **pthdb_cond_pshared** function is used to get the condition variable process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

Parameters

Item	Description
addr	Condition variable address

Item	Description
cond	Condition variable handle
mutexp	Pointer to mutex
psharedp	Pointer to pshared value
session	Session handle.

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_COND	Invalid cond handle.
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INVALID_MUTEX	Invalid mutex.
PTHDB_INTERNAL	Error in library.
PTHDB_POINTER	Invalid pointer

pthdb_mutexattr_addr, pthdb_mutexattr_prioceiling, pthdb_mutexattr_protocol, pthdb_mutexattr_pshared or pthdb_mutexattr_type Subroutine

Purpose

Gets the mutex attribute pshared, priority ceiling, protocol, and type values.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_mutexattr_addr (pthdb_session_t    session,
                          pthdb_mutexattr_t  mutexattr,
                          pthdb_addr_t  * addrp)

int pthdb_mutexattr_protocol (pthdb_session_t    session,
                              pthdb_mutexattr_t  mutexattr,
                              pthdb_protocol_t  * protocolp)

int pthdb_mutexattr_pshared (pthdb_session_t    session,
                             pthdb_mutexattr_t  mutexattr,
                             pthdb_pshared_t   * psharedp)
```

```
int pthdb_mutexattr_type (pthdb_session_t     session,
                          pthdb_mutexattr_t   mutexattr,
                          pthdb_mutex_type_t * typep)
```

Description

The **pthdb_mutexattr_addr** function reports the address of the pthread_mutexatt_t.

The **pthdb_mutexattr_prioceiling** function is used to get the mutex attribute priority ceiling value.

The **pthdb_mutexattr_protocol** function is used to get the mutex attribute protocol value. The protocol value can be **MP_INHERIT**, **MP_PROTECT**, **MP_NONE**, or **MP_NOTSUP**.

The **pthdb_mutexattr_pshared** function is used to get the mutex attribute process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

The **pthdb_mutexattr_type** is used to get the value of the mutex attribute type. The values for the mutex type can be **MK_NONRECURSIVE_NP**, **MK_RECURSIVE_NP**, **MK_FAST_NP**, **MK_ERRORCHECK**, **MK_RECURSIVE**, **MK_NORMAL**, or **MK_NOTSUP**.

Parameters

Item	Description
addr	Mutex attribute address.
mutexattr	Condition variable attribute handle
prioceiling	Pointer to priority ceiling value.
protocolp	Pointer to protocol value.
psharedp	Pointer to pshared value.
session	Session handle.
typep	Pointer to type value.

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_MUTEXATTR	Invalid mutex attribute handle.
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INTERNAL	Error in library.
PTHDB_NOSYS	Not implemented
PTHDB_POINTER	Invalid pointer

pthdb_mutex_addr, pthdb_mutex_lock_count, pthdb_mutex_owner, pthdb_mutex_pshared,

pthdb_mutex_prioceiling, pthdb_mutex_protocol, pthdb_mutex_state or pthdb_mutex_type Subroutine

Purpose

Gets the owner's pthread, mutex's pshared value, priority ceiling, protocol, lock state, and type.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_mutex_addr (pthdb_session_t  session,
                      pthdb_mutex_t    mutex,
                      pthdb_addr_t * addrp)

int pthdb_mutex_owner (pthdb_session_t  session,
                       pthdb_mutex_t    mutex,
                       pthdb_pthread_t * ownerp)

int pthdb_mutex_lock_count (pthdb_session_t  session,
                            pthdb_mutex_t    mutex,
                            int * countp);

int pthdb_mutex_pshared (pthdb_session_t  session,
                         pthdb_mutex_t    mutex,
                         pthdb_pshared_t * psharedp)

int pthdb_mutex_prioceiling (pthdb_session_t  session,
                             pthdb_mutex_t    mutex,
                             pthdb_pshared_t * prioceilingp)

int pthdb_mutex_protocol (pthdb_session_t  session,
                          pthdb_mutex_t    mutex,
                          pthdb_pshared_t * protocolp)

int pthdb_mutex_state (pthdb_session_t      session,
                       pthdb_mutex_t        mutex,
                       pthdb_mutex_state_t * statep)

int pthdb_mutex_type (pthdb_session_t     session,
                      pthdb_mutex_t       mutex,
                      pthdb_mutex_type_t * typep)
```

Description

pthdb_mutex_addr reports the address of the prhread_mutex_t.

pthdb_mutex_lock_count reports the lock count of the mutex.

pthdb_mutex_owner is used to get the pthread that owns the mutex.

The **pthdb_mutex_pshared** function is used to get the mutex process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

pthdb_mutex_prioceiling function is used to get the mutex priority ceiling value.

pthdb_mutex_protocol function is used to get the mutex protocol value. The protocol value can be **MP_INHERIT**, **MP_PROTECT**, **MP_NONE**, or **MP_NOTSUP**.

pthdb_mutex_state is used to get the value of the mutex lock state. The state can be **MS_LOCKED**, **MS_UNLOCKED** or **MS_NOTSUP**.

pthdb_mutex_type is used to get the value of the mutex type. The values for the mutex type can be **MK_NONRECURSIVE_NP**, **MK_RECURSIVE_NP**, **MK_FAST_NP**, **MK_ERRORCHECK**, **MK_RECURSIVE**, **MK_NORMAL**, or **MK_NOTSUP**.

Parameters

Item	Description
addr	Mutex address
countp	Mutex lock count
mutex	Mutex handle
ownerp	Pointer to mutex owner
psharedp	Pointer to pshared value
prioceilingp	Pointer to priority ceiling value
protocolp	Pointer to protocol value
session	Session handle.
statep	Pointer to mutex state
typep	Pointer to mutex type

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_MUTEX	Invalid mutex handle.
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INTERNAL	Call failed.
PTHDB_NOSYS	Not implemented
PTHDB_POINTER	Invalid pointer

pthdb_mutex_waiter, pthdb_cond_waiter, pthdb_rwlock_read_waiter or pthdb_rwlock_write_waiter Subroutine

Purpose

Gets the next waiter in the list of an object's waiters.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_mutex_waiter (pthdb_session_t  session,
                        pthdb_mutex_t    mutex,
                        pthdb_pthread_t * waiter,
                        int              cmd);
int  pthdb_cond_waiter (pthdb_session_t  session,
                        pthdb_cond_t     cond,
                        pthdb_pthread_t * waiter,
                        int              cmd)
int *pthdb_rwlock_read_waiter (pthdb_session_t  session,
                               pthdb_rwlock_t   rwlock,
                               pthdb_pthread_t * waiter,
                               int              cmd)
int *pthdb_rwlock_write_waiter (pthdb_session_t  session,
                                pthdb_rwlock_t   rwlock,
                                pthdb_pthread_t * waiter,
                                int              cmd)
```

Description

The **pthdb_mutex_waiter** functions get the next waiter in the list of an object's waiters.

Each list is reset to the top of the list when the **pthdb_session_update** function is called, or when the list function reports a **PTHDB_INVALID_*** value. For example, when **pthdb_attr** reports an attribute of **PTHDB_INVALID_ATTR** the list is reset to the beginning such that the next call reports the first attribute in the list, if any.

A report of **PTHDB_INVALID_***OBJECT* represents the empty list or the end of a list, where *OBJECT* is one of these values: **PTHREAD**, **ATTR**, **MUTEX**, **MUTEXATTR**, **COND**, **CONDATTR**, **RWLOCK**, **RWLOCKATTR**, **KEY**, or **TID** as appropriate.

When **PTHDB_LIST_FIRST** is passed for the *cmd* parameter, the first item in the list is retrieved.

Parameters

Item	Description
session	Session handle.
mutex	Mutex object.
cond	Condition variable object.
cmd	Reset to the beginning of the list.
rwlock	Read/Write lock object.
waiter	Pointer to waiter.

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_CMD	Invalid command.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INTERNAL	Error in library.
PTHDB_MEMORY	Not enough memory
PTHDB_POINTER	Invalid pointer

pthdb_pthread_arg Subroutine

Purpose

Reports the information associated with a pthread.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_pthread_arg (pthdb_session_t  session,
                       pthdb_pthread_t  pthread,
                       pthdb_addr_t    * argp)

int pthdb_pthread_addr (pthdb_session_t  session,
                        pthdb_pthread_t  pthread,
                        pthdb_addr_t    *addrp)

int pthdb_pthread_cancelpend (pthdb_session_t  session,
                              pthdb_pthread_t  pthread,
                              int              * cancelpendp)

int pthdb_pthread_cancelstate (pthdb_session_t      session,
                               pthdb_pthread_t      pthread,
                               pthdb_cancelstate_t * cancelstatep)

int pthdb_pthread_canceltype (pthdb_session_t      session,
                              pthdb_pthread_t      pthread,
                              pthdb_canceltype_t * canceltypep)

int pthdb_pthread_detachstate (pthdb_session_t  session,
                               pthdb_pthread_t  pthread,
                               pthdb_detachstate_t * detachstatep)

int pthdb_pthread_exit (pthdb_session_t  session,
                        pthdb_pthread_t  pthread,
                        pthdb_addr_t    * exitp)

int pthdb_pthread_func (pthdb_session_t  session,
                        pthdb_pthread_t  pthread,
                        pthdb_addr_t    * funcp)
```

```
int pthdb_pthread_ptid (pthdb_session_t  session,
                        pthdb_pthread_t  pthread,
                        pthread_t       * ptidp)

int pthdb_pthread_schedparam (pthdb_session_t     session,
                              pthdb_pthread_t     pthread,
                              struct sched_param * schedparamp);

int pthdb_pthread_schedpolicy (pthdb_session_t  session,
                               pthdb_pthread_t  pthread,
                               pthdb_schedpolicy_t  * schedpolicyp)

int pthdb_pthread_schedpriority (pthdb_session_t  session,
                                 pthdb_pthread_t  pthread,
                                 int             * schedprioritytp)

int pthdb_pthread_scope (pthdb_session_t  session,
                         pthdb_pthread_t  pthread,
                         pthdb_scope_t   * scopep)

int pthdb_pthread_state (pthdb_session_t  session,
                         pthdb_pthread_t  pthread,
                         pthdb_state_t  * statep)

int pthdb_pthread_suspendstate (pthdb_session_t  session,
                                pthdb_pthread_t  pthread,
                                pthdb_suspendstate_t * suspendstatep)

int pthdb_ptid_pthread (pthdb_session_t  session,
                        pthread_t  ptid,
                        pthdb_pthread_t    * pthreadp)
```

Description

pthdb_pthread_arg reports the initial argument passed to the pthread's start function.

pthdb_pthread_addr reports the address of the pthread_t.

pthdb_pthread_cancelpend reports non-zero if cancellation is pending on the pthread; if not, it reports zero.

pthdb_pthread_cancelstate reports whether cancellation is enabled (**PCS_ENABLE**) or disabled (**PCS_DISABLE**). **PCS_NOTSUP** is reserved for unexpected results.

pthdb_pthread_canceltype reports whether cancellation is deferred (**PCT_DEFERRED**) or asynchronous (**PCT_ASYNCHRONOUS**). **PCT_NOTSUP** is reserved for unexpected results.

pthdb_pthread_detachstate reports whether the pthread is detached (**PDS_DETACHED**) or joinable (**PDS_JOINABLE**). **PDS_NOTSUP** is reserved for unexpected results.

pthdb_pthread_exit reports the exit status returned by the pthread via **pthread_exit**. This is only valid if the pthread has exited (**PST_TERM**).

pthdb_pthread_func reports the address of the pthread's start function.

pthdb_pthread_ptid reports the pthread identifier (**pthread_t**) associated with the pthread.

pthdb_pthread_schedparam reports the pthread's scheduling parameters. This currently includes policy and priority.

pthdb_pthread_schedpolicy reports whether the pthread's scheduling policy is other (**SP_OTHER**), first in first out (**SP_FIFO**), or round robin (**SP_RR**). **SP_NOTSUP** is reserved for unexpected results.

pthdb_pthread_schedpriority reports the pthread's scheduling priority.

pthdb_pthread_scope reports whether the pthread has process scope (**PS_PROCESS**) or system scope (**PS_SYSTEM**). **PS_NOTSUP** is reserved for unexpected results.

pthdb_pthread_state reports whether the pthread is being created (**PST_IDLE**), currently running (**PST_RUN**), waiting on an event (**PST_SLEEP**), waiting on a cpu (**PST_READY**), or waiting on a join or detach (**PST_TERM**). **PST_NOTSUP** is reserved for unexpected results.

pthdb_pthread_suspendstate reports whether the pthread is suspended (**PSS_SUSPENDED**) or not (**PSS_UNSUSPENDED**). **PSS_NOTSUP** is reserved for unexpected results.

pthdb_ptid_pthread reports the pthread for the ptid.

Parameters

Item	Description
addr	pthread address
argp	Initial argument buffer.
cancelpendp	Cancel pending buffer.
cancelstatep	Cancel state buffer.
canceltypep	Cancel type buffer.
detachstatep	Detach state buffer.
exitp	Exit value buffer.
funcp	Start function buffer.
pthread	pthread handle.
pthreadp	Pointer to pthread handle.
ptid	pthread identifier
ptidp	pthread identifier buffer.
schedparamp	Scheduling parameters buffer.
schedpolicyp	Scheduling policy buffer.
schedpriorityp	Scheduling priority buffer.
scopep	Contention scope buffer.
session	Session handle.
statep	State buffer.
suspendstatep	Suspend state buffer.

Return Values

If successful, these functions return **PTHDB_SUCCESS**, else an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_PTHREAD	Invalid pthread handle.
PTHDB_BAD_POINTER	Invalid buffer pointer.
PTHDB_BAD_PTID	Invalid ptid.
PTHDB_CALLBACK	Debugger call back error.

Item	Description
PTHDB_NOTSUP	Not supported.
PTHDB_INTERNAL	Error in library.

pthdb_pthread_context or pthdb_pthread_setcontext Subroutine

Purpose

Provides access to the pthread context via the *struct context64* structure.

Library

pthread debug library (**libpthdebug.a)**

Syntax

```
#include <sys/pthdebug.h>

int pthdb_pthread_context (pthdb_session_t   session,
                           pthdb_pthread_t   pthread,
                           pthdb_context_t * context)

int pthdb_pthread_setcontext (pthdb_session_t   session,
                              pthdb_pthread_t   pthread,
                              pthdb_context_t * context)
```

Description

The pthread debug library provides access to the pthread context via the *struct context64* structure, whether the process is 32-bit or 64-bit. The debugger should be able to convert from 32-bit to 64-bit and from 64-bit for 32-bit processes. The extent to which this structure is filled in depends on the presence of the **PTHDB_FLAG_GPRS**, **PTHDB_FLAG_SPRS**l and **PTHDB_FLAG_FPRS** session flags. It is necessary to use the pthread debug library to access the context of a pthread without a kernel thread. The pthread debug library can also be used to access the context of a pthread with a kernel thread, but this results in a call back to the debugger, meaning that the debugger is capable of obtaining this information by itself. The debugger determines if the kernel thread is running in user mode or kernel mode and then fills in the *struct context64* appropriately. The pthread debug library does not use this information itself and is thus not sensitive to the correct implementation of the **read_regs** and **write_regs** call back functions.

pthdb_pthread_context reports the context of the pthread based on the settings of the session flags. Uses the **read_regs** call back if the pthread has a kernel thread. If **read_regs** is not defined, then it returns **PTHDB_NOTSUP**.

pthdb_pthread_setcontext sets the context of the pthread based on the settings of the session flags. Uses the **write_data** call back if the pthread does not have a kernel thread. Use the **write_regs** call back if the pthread has a kernel thread.

If the debugger does not define the **read_regs** and **write_regs** call backs and if the pthread does not have a kernel thread, then the **pthdb_pthread_context** and **pthdb_pthread_setcontext** functions succeed. But if a pthread does not have a kernel thread, then these functions fail and return **PTHDB_CONTEXT**.

Parameters

Item	Description
session	Session handle.

Item	Description
pthread	pthread handle.
context	Context buffer pointer.

Return Values

If successful, these functions return *PTHDB_SUCCESS*. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_PTHREAD	Invalid pthread handle.
PTHDB_BAD_POINTER	Invalid buffer pointer.
PTHDB_CALLBACK	Callback function failed.
PTHDB_CONTEXT	Could not determine pthread context.
PTHDB_MEMORY	Not enough memory
PTHDB_NOTSUP	**pthdb_pthread_(set)context** returns **PTHDB_NOTSUP** if the **read_regs**, **write_data** or **write_regs** call backs are set to NULL.
PTHDB_INTERNAL	Error in library.

pthdb_pthread_hold, pthdb_pthread_holdstate or pthdb_pthread_unhold Subroutine

Purpose

Reports and changes the hold state of the specified pthread.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_pthread_holdstate (pthdb_session_t    session,
                             pthdb_pthread_t    pthread,
                             pthdb_holdstate_t * holdstatep)
int pthdb_pthread_hold (pthdb_session_t  session,
                        pthdb_pthread_t  pthread)
int pthdb_pthread_unhold (pthdb_session_t  session,
                          pthdb_pthread_t  pthread)
```

Description

pthdb_pthread_holdstate reports if a pthread is held. The possible hold states are **PHS_HELD**, **PHS_NOTHELD**, or **PHS_NOTSUP**.

pthdb_pthread_hold prevents the specified pthread from running.

pthdb_pthread_unhold unholds the specified pthread. The pthread held earlier can be unheld by calling this function.

Note:

1. You must always use the **pthdb_pthread_hold** and **pthdb_pthread_unhold** functions, regardless of whether or not a pthread has a kernel thread.
2. These functions are only supposted when the **PTHDB_FLAG_HOLD** is set.

Parameters

Item	Description
session	Session handle.
pthread	pthread handle. The specified pthread should have an attached kernel thread id.
holdstatep	Pointer to the hold state

Return Values

If successful, **pthdb_pthread_hold** returns **PTHDB_SUCCESS**. Otherwise, it returns an error code.

Error Codes

Item	Description
PTHDB_BAD_PTHREAD	Invalid pthread handle.
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_HELD	pthread is held.
PTHDB_INTERNAL	Error in library.

pthdb_pthread_sigmask, pthdb_pthread_sigpend or pthdb_pthread_sigwait Subroutine

Purpose

Returns the pthread signals pending, the signals blocked, the signals received, and awaited signals.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_pthread_sigmask (pthdb_session_t  session,
                           pthdb_pthread_t  pthread,
                           sigset_t        * sigsetp)
int pthdb_pthread_sigpend (pthdb_session_t  session,
                           pthdb_pthread_t  pthread,
                           sigset_t        * sigsetp)
int pthdb_pthread_sigwait (pthdb_session_t  session,
                           pthdb_pthread_t   pthread,
                           sigset_t        * sigsetp)
```

Description

pthdb_pthread_sigmask reports the signals that the pthread has blocked.

pthdb_pthread_sigpend reports the signals that the pthread has pending.

pthdb_pthread_sigwait reports the signals that the pthread is waiting on.

Parameters

Item	Description
session	Session handle.
pthread	Pthread handle
sigsetp	Signal set buffer.

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Code

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_PTHREAD	Invalid pthread handle.
PTHDB_BAD_POINTER	Invalid buffer pointer.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INTERNAL	Error in library.

pthdb_pthread_specific Subroutine

Purpose

Reports the value associated with a pthreads specific data key.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

void *pthdb_pthread_specific(pthdb_session_t  session,
                             pthdb_pthread_t  pthread,
                             pthdb_key_t    key,
                             pthdb_addr_t    * specificp)
```

Description

Each process has active pthread specific data keys. Each active pthread specific data key is in use by one or more pthreads. Each pthread can have its own value associated with each pthread specific data key. The **pthdb_pthread_specific** function provide access to those values.

pthdb_pthread_specific reports the specific data value for the pthread and key combination.

Parameters

Item	Description
session	The session handle.
pthread	The pthread handle.
key	The key.
specificp	Specific data value buffer.a

Return Values

If successful, **pthdb_pthread_specific** returns **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_PTHREAD	Invalid pthread handle.
PTHDB_BAD_KEY	Invalid key.
PTHDB_BAD_POINTER	Invalid buffer pointer.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INTERNAL	Error in library.

pthdb_pthread_tid or pthdb_tid_pthread Subroutine

Purpose

Gets the kernel thread associated with the pthread and the pthread associated with the kernel thread.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_pthread_tid (pthdb_session_t  session,
                       pthdb_pthread_t  pthread,
                       tid_t           * tidp)
int pthdb_tid_pthread (pthdb_session_t  session,
                       tid_t            tid,
                       pthdb_pthread_t * pthreadp)
```

Description

pthdb_pthread_tid gets the kernel thread id associated with the pthread.

pthdb_tid_pthread is used to get the pthread associated with the kernel thread.

Parameters

Item	Description
session	Session handle.
pthread	Pthread handle
pthreadp	Pointer to pthread handle
tid	Kernel thread id
tidp	Pointer to kernel thread id

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_PTHREAD	Invalid pthread handle.
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_TID	Invalid *tid*.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INTERNAL	Error in library.
PTHDB_INVALID_TID	Empty list or the end of a list.

pthdb_rwlockattr_addr, or pthdb_rwlockattr_pshared Subroutine

Purpose

Gets the rwlock attribute pshared values.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_rwlockattr_addr (pthdb_session_t      session,
                           pthdb_rwlockattr_t  rwlockattr,
                           pthdb_addr_t    * addrp)

int pthdb_rwlockattr_pshared (pthdb_session_t     session,
                              pthdb_rwlockattr_t  rwlockattr,
                              pthdb_pshared_t    * psharedp)
```

Description

pthdb_rwlockattr_addr reports the address of the pthread_rwlockattr_t.

pthdb_rwlockattr_pshared is used to get the rwlock attribute process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

Parameters

Item	Description
addr	Read/Write lock attribute address.
psharedp	Pointer to the pshared value.
rwlockattr	Read/Write lock attribute handle
session	Session handle.

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_RWLOCKATTR	Invalid rwlock attribute handle.
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_INTERNAL	Error in library.
PTHDB_POINTER	Invalid pointer

Related Information

The **pthdebug.h** file.

The **pthread.h** file.

pthdb_rwlock_addr, pthdb_rwlock_lock_count, pthdb_rwlock_owner, pthdb_rwlock_pshared or pthdb_rwlock_state Subroutine

Purpose

Gets the owner, the pshared value, or the state of the read/write lock.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_rwlock_addr (pthdb_session_t  session,
                       pthdb_rwlock_t   rwlock,
                       pthdb_addr_t * addrp)

int pthdb_rwlock_lock_count (pthdb_session_t  session,
                             pthdb_rwlock_t   rwlock,
                             int * countp);
```

```
int pthdb_rwlock_owner (pthdb_session_t  session,
                        pthdb_rwlock_t   rwlock,
                        pthdb_pthread_t * ownerp
                        int              cmd)

int pthdb_rwlock_pshared (pthdb_session_t  session,
                          pthdb_rwlock_t   rwlock,
                          pthdb_pshared_t * psharedp)

int pthdb_rwlock_state (pthdb_session_t       session,
                        pthdb_rwlock_t        rwlock,
                        pthdb_rwlock_state_t * statep)
```

Description

The **pthdb_rwlock_addr** function reports the address of the pthdb_rwlock_t.

The **pthdb_rwlock_lock_count** function reports the lock count for the rwlock.

The **pthdb_rwlock_owner** function is used to get the read/write lock owner's pthread handle.

The **pthdb_rwlock_pshared** function is used to get the rwlock attribute process shared value. The pshared value can be **PSH_SHARED**, **PSH_PRIVATE**, or **PSH_NOTSUP**.

The **pthdb_rwlock_state** is used to get the read/write locks state. The state can be **RWLS_NOTSUP**, **RWLS_WRITE**, **RWLS_FREE**, and **RWLS_READ**.

Parameters

Item	Description
addrp	Read write lock address.
countp	Read write lock lock count.
cmd	*cmd* can be **PTHDB_LIST_FIRST** to get the first owner in the list of owners or **PTHDB_LIST_NEXT** to get the next owner in the list of owners. The list is empty or ended by ***owner == PTHDB_INVALID_PTHREAD**.
ownerp	Pointer to pthread which owns the rwlock
psharedp	Pointer to pshared value
rwlock	Read write lock handle
session	Session handle.
statep	Pointer to state value

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, an error code is returned.

Error Codes

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_CMD	Invalid command passed.
PTHDB_CALLBACK	Debugger call back error.

Item	Description
PTHDB_INTERNAL	Error in library.
PTHDB_POINTER	Invalid pointer

pthdb_session_committed Subroutines

Purpose

Facilitates examining and modifying multi-threaded application's pthread library object data.

Library

pthread debug library (**libpthdebug.a**)

Syntax

```
#include <sys/pthdebug.h>

int pthdb_session_committed (pthdb_session_t  session,
                             char           ** name);
int pthdb_session_concurrency (pthdb_session_t  session,
                               int            * concurrencyp);
int pthdb_session_destroy (pthdb_session_t  session)
int pthdb_session_flags (pthdb_session_t session,
                         unsigned long long * flagsp)
int pthdb_session_init (pthdb_user_t  user,
                        pthdb_exec_mode_t  exec_mode,
                        unsigned long long  flags,
                        pthdb_callbacks_t * callbacks,
                        pthdb_session_t * sessionp)
int pthdb_session_pthreaded (pthdb_user_t   user,
                             unsigned long long  flags
                             pthdb_callbacks_t * callbacks,
                             char             ** name)
int pthdb_session_continue_tid (pthdb_session_t  session,
                                tid_t           * tidp,
                                int              cmd);
int pthdb_session_stop_tid (pthdb_session_t  session,
                            tid_t           tid);
int pthdb_session_commit_tid (pthdb_session_t  session,
                              tid_t           * tidp,
                              int               cmd);
int pthdb_session_setflags (pthdb_session_t     session,
                            unsigned long long  flags)
int pthdb_session_update (pthdb_session_t  session)
```

Description

To facilitate debugging multiple processes, the pthread debug library supports multiple sessions, one per process. Functions are provided to initialize, destroy, and customize the behavior of these sessions. In addition, functions are provided to query global fields of the pthread library. All functions in the library require a session handle associated with an initialized session except **pthdb_session_init**, which initializes sessions, and **pthdb_session_pthreaded**, which can be called before the session has been initialized.

pthdb_session_committed reports the symbol name of a function called after the hold/unhold commit operation has completed. This symbol name can be used to set a breakpoint to notify the debugger when the hold/unhold commit has completed. The actual symbol name reported may change at any time. The function name returned is implemented in assembly with the following code:

```
                ori 0,0, 0          # no-op
                blr                 # return to caller
```

This allows the debugger to overwrite the no-op with a trap instruction and leave it there by stepping over it. This function is only supported when the **PTHDB_FLAG_HOLD** flag is set.

pthdb_session_concurrency reports the concurrency level of the pthread library. The concurrency level is the M:N ratio, where N is always 1.

pthdb_session_destroy notifies the pthread debug library that the debugger or application is finished with the session. This deallocates any memory associated with the session and allows the session handle to be reused.

pthdb_session_setflags changes the flags for a session. With these flags, a debugger can customize the session. Flags consist of the following values or-ed together:

Item	Description
PTHDB_FLAG_GPRS	The general purpose registers should be included in any context read or write, whether internal to the library or via call backs to the debugger.
PTHDB_FLAG_SPRS	The special purpose registers should be included in any context read or write whether internal to the library or via call backs to the debugger.
PTHDB_FLAG_FPRS	The floating point registers should be included in any context read or write whether internal to the library or via call backs to the debugger.
PTHDB_FLAG_REGS	All registers should be included in any context read or write whether internal to the library or via call backs to the debugger. This is equivalent to **PTHDB_FLAG_GPRS| PTHDB_FLAG_GPRS|PTHDB_FLAG_GPRS**.
PTHDB_FLAG_HOLD	The debugger will be using the pthread debug library hold/unhold facilities to prevent the execution of pthreads. This flag cannot be used with **PTHDB_FLAG_SUSPEND**. This flag should be used by debuggers, only.
PTHDB_FLAG_SUSPEND	Applications will be using the pthread library suspend/continue facilities to prevent the execution of pthreads. This flag cannot be used with **PTHDB_FLAG_HOLD**. This flag is for introspective mode and should be used by applications, only. **Note: PTHDB_FLAG_HOLD** and **PTHDB_FLAG_SUSPEND** can only be passed to the **pthdb_session_init** function. Neither **PTHDB_FLAG_HOLD** nor **PTHDB_FLAG_SUSPEND** should be passed to **pthdb_session_init** when debugging a core file.

The **pthdb_session_flags** function gets the current flags for the session.

The **pthdb_session_init** function tells the pthread debug library to initialize a session associated with the unique given user handle. **pthdb_session_init** will assign a unique session handle and return it to the debugger. If the application's execution mode is 32 bit, then the debugger should initialize the **exec_mode** to **PEM_32BIT**. If the application's execution mode is 64 bit, then the debugger should initialize **mode** to **PEM_64BIT**. The **flags** are documented above with the **pthdb_session_setflags** function. The **callback** parameter is a list of call back functions. (Also see the **pthdebug.h** header file.) The **pthdb_session_init** function calls the **symbol_addrs** function to get the starting addresses of the symbols and initializes these symbols' starting addresses within the pthread debug library.

pthdb_session_pthreaded reports the symbol name of a function called after the pthread library has been initialized. This symbol name can be used to set a breakpoint to notify the debugger when to initialize a pthread debug library session and begin using the pthread debug library to examine pthread library state. The actual symbol name reported may change at any time. This function, is the only pthread debug library function that can be called before the pthread library is initialized. The function name returned is implemented in assembly with the following code:

```
ori 0,0,0           # no-op
blr                 # return to caller
```

This is conveniently allows the debugger to overwrite the no-op with a trap instruction and leave it there by stepping over it.

The **pthdb_session_continue_tid** function allows the debugger to obtain the list of threads that must be continued before it proceeds with single stepping a single pthread or continuing a group of pthreads. This function reports one tid at a time. If the list is empty or the end of the list has been reached, **PTHDB_INVALID_TID** is reported. The debugger will need to continue any pthreads with kernel threads that it wants. The debugger is responsible for parking the stop thread and continuing the stop thread. The *cmd* parameter can be either **PTHDB_LIST_NEXT** or **PTHDB_LIST_FIRST**; if **PTHDB_LIST_FIRST** is passed, then the internal counter will be reset and the first tid in the list will be reported.

Note: This function is only supported when the **PTHDB_FLAG_HOLD** flag is set.

The **pthdb_session_stop_tid** function informs the pthread debug library, which informs the pthread library the tid of the thread that stopped the debugger.

Note: This function is only supported when the **PTHDB_FLAG_HOLD** flag is set.

pthdb_session_commit_tid reports subsequent kernel thread identifiers which must be continued to commit the hold and unhold changes. This function reports one tid at a time. If the list is empty or the end of the list has been reached, **PTHDB_INVALID_TID** is reported. The *cmd* parameter can be either **PTHDB_LIST_NEXT** or **PTHDB_LIST_FIRST**, if **PTHDB_LIST_FIRST** is passed then the internal counter will be reset and first tid in the list will be reported.

Note: This function is only supported when the **PTHDB_FLAG_HOLD** flag is set.

pthdb_session_update tells the pthread debug library to update it's internal information concerning the state of the pthread library. This should be called each time the process stops before any other pthread debug library functions to ensure their results are reliable.

Each list is reset to the top of the list when the **pthdb_session_update** function is called, or when the list function reports a **PTHDB_INVALID_*** value. For example, when **pthdb_attr** reports an attribute of **PTHDB_INVALID_ATTR** the list is reset to the beginning such that the next call reports the first attribute in the list, if any.

A report of **PTHDB_INVALID_***OBJECT* represents the empty list or the end of a list, where *OBJECT* is one of these values: **PTHREAD**, **ATTR**, **MUTEX**, **MUTEXATTR**, **COND**, **CONDATTR**, **RWLOCK**, **RWLOCKATTR**, **KEY**, or **TID** as appropriate.

Parameters

Item	Description
session	Session handle.
user	Debugger user handle.
sessionp	Pointer to session handle.
name	Symbol name buffer.
cmd	Reset to the beginning of the list.
concurrencyp	Library concurrency buffer.
flags	Session flags.
flagsp	Pointer to session flags.
exec_mode	Debuggee execution mode: **PEM_32BIT** for 32-bit processes or **PEM_64BIT** for 64-bit processes.
callbacks	Call backs structure.
tid	Kernel thread id.
tidp	Kernel thread id buffer..

Return Values

If successful, these functions return **PTHDB_SUCCESS**. Otherwise, they return an error value.

Error Codes

Item	Description
PTHDB_BAD_SESSION	Invalid session handle.
PTHDB_BAD_VERSION	Invalid pthread debug library or pthread library version.
PTHDB_BAD_MODE	Invalid execution mode.
PTHDB_BAD_FLAGS	Invalid session flags.

Item	Description
PTHDB_BAD_CALLBACK	Insufficient call back functions.
PTHDB_BAD_CMD	Invalid command.
PTHDB_BAD_POINTER	Invalid buffer pointer.
PTHDB_BAD_USER	Invalid user handle.
PTHDB_CALLBACK	Debugger call back error.
PTHDB_MEMORY	Not enough memory.
PTHDB_NOSYS	Function not implemented.
PTHDB_NOT_PTHREADED	pthread library not initialized.
PTHDB_SYMBOL	pthread library symbol not found.
PTHDB_INTERNAL	Error in library.

pthread_atfork Subroutine

Purpose

Registers fork handlers.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <sys/types.h>
#include <unistd.h>

int pthread_atfork (prepare, parent, child)
void (*prepare)(void);
void (*parent)(void);
void (*child)(void);
```

Description

The **pthread_atfork** subroutine registers fork cleanup handlers. The *prepare* handler is called before the processing of the **fork** subroutine commences. The *parent* handler is called after the processing of the **fork** subroutine completes in the parent process. The *child* handler is called after the processing of the **fork** subroutine completes in the child process.

When the **fork** subroutine is called, only the calling thread is duplicated in the child process, but all synchronization variables are duplicated. The **pthread_atfork** subroutine provides a way to prevent state inconsistencies and resulting deadlocks. The expected usage is that the *prepare* handler acquires all mutexes, and the two other handlers release them in the parent and child processes.

The prepare handlers are called in LIFO (Last In First Out) order; whereas the parent and child handlers are called in FIFO (first-in first-out) order. Thereafter, the order of calls to the **pthread_atfork** subroutine is significant.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library.

Parameters

Item	Description
prepare	Points to the pre-fork cleanup handler. If no pre-fork handling is desired, the value of this pointer should be set to **NULL**.

Item	Description
parent	Points to the parent post-fork cleanup handler. If no parent post-fork handling is desired, the value of this pointer should be set to **NULL**.
child	Points to the child post-fork cleanup handler. If no child post-fork handling is desired, the value of this pointer should be set to **NULL**.

Return Values

Upon successful completion, the **pthread_atfork** subroutine returns a value of zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_atfork** subroutine will fail if:

Item	Description
ENOMEM	Insufficient table space exists to record the fork handler addresses.

The **pthread_atfork** subroutine will not return an error code of **EINTR**.

pthread_atfork_np subroutine`

Purpose

Registers fork handlers.

Library

Threads Library (`libpthreads.a`)

Syntax

```
#include <sys/types.h>
#include <unistd.h>

int pthread_atfork_np (arg, prepare, parent, child)
void *arg;
void (*prepare)(void *);
void (*parent)(void *);
void (*child)(void *);
```

Description

The pthread_atfork_np subroutine registers cleanup handlers for the fork subroutine. The arg is the parameter to be passed to the functions for pre and post fork handling. The prepare handler is called before the processing of the fork subroutine commences. The parent handler is called after the processing of the fork subroutine completes in the parent process. The child handler is called after the processing of the fork subroutine completes in the child process.

When the fork subroutine is called, only the calling thread is duplicated in the child process, but all synchronization variables are duplicated. The pthread_atfork_np subroutine provides a way to prevent state inconsistencies and resulting deadlocks. The expected usage is that the prepare handler acquires all mutexes, and the two other handlers release them in the parent and child processes.

The prepare handlers are called in LIFO (Last In First Out) order; whereas the parent and child handlers are called in FIFO (first-in first-out) order. Therefore, the order of calls to the pthread_atfork_np subroutine is significant.

Note:

- The pthread.h header file must be the first included file of each source file using the threads library.
- The pthread_atfork_np subroutine is not portable.

Paramaters

arg
Points to the parameter to be passed to the fork cleanup handlers.

prepare
The pre-fork cleanup handler. If no pre-fork handling is desired, the value of this pointer should be set to NULL.

parent
The parent post-fork cleanup handler. If no parent post-fork handling is desired, the value of this pointer should be set to NULL.

child
The child post-fork cleanup handler. If no child post-fork handling is desired, the value of this pointer should be set to NULL.

Return Values

Upon successful completion, the pthread_atfork_np subroutine returns a value of zero. Otherwise, an error number is returned to indicate the error

Error Codes

ENOMEM
Insufficient table space exists to record the fork handler addresses.

pthread_atfork_unregister_np Subroutine`

Purpose
Unregisters fork handlers.

Library

Threads Library (libpthreads.a).

Syntax

```
#include <sys/types.h>
#include <unistd.h>
int pthread_atfork_unregister_np (arg, prepare, parent, child, flags)
void *arg;
void (*prepare)();
void (*parent)();
void (*child)();
int flags;
```

Description

The pthread_atfork_unregister_np subroutine unregisters functions for pre and post fork handling. The fork handlers must be previously registered using either the pthread_atfork or the pthread_atfork_np subroutine.

The flags parameter determines what handlers are unregistered. It could be any of the following :

0
The first POSIX handler that matches will be unregistered.

PTHREAD_ATFORK_ALL
All POSIX duplicate handlers that match and all non- portable handlers that differ only in argument value will be unregistered.

PTHREAD_ATFORK_ARGUMENT
The first non-portable handler that matches will be unregistered.

The above flags may be combined using the bitwise OR operation. , The flags value of PTHREAD_ATFORK_ARGUMENT | PTHREAD_ATFORK_ALL would cause all non-portable duplicate handlers that match to be unregistered.

Note:

- The pthread.h header file must be the first included file of each source file using the threads library.
- The pthread_atfork_unregister_np subroutine is not portable.
- The handlers that take parameter are non-portable.
- The handlers that do not take parameter are POSIX compliant and are referred to as POSIX handlers.

Paramaters

arg
Points to the parameter to be passed to the fork cleanup handlers. If the handlers do not take parameter , the value of this pointer should be set to NULL.

prepare
The pre-fork cleanup handler.

parent
The parent post-fork cleanup handler.

child
The child post-fork cleanup handler.

flags
Defines what handlers are to be unregistered.

Return Values

Upon successful completion, the pthread_atfork_unregister_np subroutine returns a value of zero. Otherwise, an error number is returned to indicate the error.

Error Codes

EINVAL
Arguments do not identify a fork handler.

pthread_attr_destroy Subroutine

Purpose

Deletes a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
int pthread_attr_destroy (attr)
pthread_attr_t *attr;
```

Description

The **pthread_attr_destroy** subroutine destroys the thread attributes object *attr*, reclaiming its storage space. It has no effect on the threads previously created with that object.

Parameters

Item	Description
attr	Specifies the thread attributes object to delete.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_destroy** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.

This function will not return an error code of [EINTR].

pthread_attr_getguardsize or pthread_attr_setguardsize Subroutines

Purpose

Gets or sets the thread guardsize attribute.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_getguardsize (attr, guardsize)
const pthread_attr_t *attr;
size_t *guardsize;

int pthread_attr_setguardsize (attr, guardsize)
pthread_attr_t *attr;
size_t guardsize;
```

Description

The *guardsize* attribute controls the size of the guard area for the created thread's stack. The *guardsize* attribute provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results (possibly in a SIGSEGV signal being delivered to the thread).

The guardsize attribute is provided to the application for two reasons:

- Overflow protection can potentially result in wasted system resources. An application that creates a large number of threads, and which knows its threads will never overflow their stack, can save system resources by turning off guard areas.
- When threads allocate large data structures on the stack, large guard areas may be needed to detect stack overflow.

The **pthread_attr_getguardsize** function gets the guardsize attribute in the *attr* object. This attribute is returned in the *guardsize* parameter.

The **pthread_attr_setguardsize** function sets the guardsize attribute in the *attr* object. The new value of this attribute is obtained from the *guardsize* parameter. If *guardsize* is zero, a guard area will not be provided for threads created with attr. If *guardsize* is greater than zero, a guard area of at least size guardsize bytes is provided for each thread created with attr.

A conforming implementation is permitted to round up the value contained in *guardsize* to a multiple of the configurable system variable PAGESIZE (see **sys/mman.h**). If an implementation rounds up the value of guardsize to a multiple of PAGESIZE, a call to **pthread_attr_getguardsize** specifying *attr* will store in the *guardsize* parameter the guard size specified by the previous **pthread_attr_setguardsize** function call. The default value of the guardsize attribute is PAGESIZE bytes. The actual value of PAGESIZE is implementation-dependent and may not be the same on all implementations.

If the stackaddr attribute has been set (that is, the caller is allocating and managing its own thread stacks), the guardsize attribute is ignored and no protection will be provided by the implementation. It is the responsibility of the application to manage stack overflow along with stack allocation and management in this case.

Parameters

Item	Description
attr	Specifies the thread attributes object.
guardsize	Controls the size of the guard area for the created thread's stack, and protects against overflow of the stack pointer.

Return Values

If successful, the **pthread_attr_getguardsize** and **pthread_attr_setguardsize** functions return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_attr_getguardsize** and **pthread_attr_setguardsize** functions will fail if:

Item	Description
EINVAL	The attribute *attr* is invalid.
EINVAL	The *guardsize* parameter is invalid.
EINVAL	The *guardsize* parameter contains an invalid value.

pthread_attr_getinheritsched, pthread_attr_setinheritsched Subroutine

Purpose

Gets and sets the **inheritsched** attribute (REALTIME THREADS).

Syntax

```
#include <pthread.h>
#include <time.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict attr,
       int *restrict inheritsched);
int pthread_attr_setinheritsched(pthread_attr_t *attr,
       int inheritsched);
```

Description

The **pthread_attr_getinheritsched()** and **pthread_attr_setinheritsched()** functions, respectively, get and set the **inheritsched** attribute in the *attr* argument.

When the attributes objects are used by **pthread_create()**, the **inheritsched** attribute determines how the other scheduling attributes of the created thread are set.

Item	Description
PTHREAD_INHERIT_SCHED	Specifies that the thread scheduling attributes is inherited from the creating thread, and the scheduling attributes in this *attr* argument are ignored.
PTHREAD_EXPLICIT_SCHED	Specifies that the thread scheduling attributes are set to the corresponding values from this attributes object.

The PTHREAD_INHERIT_SCHED and PTHREAD_EXPLICIT_SCHED symbols are defined in the **<pthread.h>** header.

The following thread scheduling attributes defined by IEEE Std 1003.1-2001 are affected by the **inheritsched** attribute: scheduling policy (**schedpolicy**), scheduling parameters (**schedparam**), and scheduling contention scope (**contentionscope**).

Application Usage

After these attributes have been set, a thread can be created with the specified attributes using **pthread_create()**. Using these routines does not affect the current running thread.

Return Values

If successful, the **pthread_attr_getinheritsched()** and **pthread_attr_setinheritsched()** functions return 0; otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_attr_setschedpolicy()** function might fail if:

Item	Description
EINVAL	The value of *inheritsched* is not valid.
ENOTSUP	An attempt was made to set the attribute to an unsupported value.

These functions do not return an error code of EINTR.

pthread_attr_getschedparam Subroutine

Purpose

Returns the value of the schedparam attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>

int pthread_attr_getschedparam (attr, schedparam)
const pthread_attr_t *attr;
struct sched_param *schedparam;
```

Description

The **pthread_attr_getschedparam** subroutine returns the value of the schedparam attribute of the thread attributes object *attr*. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The `sched_priority` field of the **sched_param** structure contains the priority of the thread. It is an integer value.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
attr	Specifies the thread attributes object.
schedparam	Points to where the schedparam attribute value will be stored.

Return Values

Upon successful completion, the value of the schedparam attribute is returned via the *schedparam* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_getschedparam** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.

This function does not return EINTR.

pthread_attr_getschedpolicy, pthread_attr_setschedpolicy Subroutine

Purpose

Gets and sets the **schedpolicy** attribute (REALTIME THREADS).

Syntax

```
#include <pthread.h>
#include <time.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict attr,
```

```
        int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

Description

The **pthread_attr_getschedpolicy()** and **pthread_attr_setschedpolicy()** functions, respectively, get and set the **schedpolicy** attribute in the *attr* argument.

The supported values of policy include SCHED_FIFO, SCHED_RR, and SCHED_OTHER, which are defined in the **<sched.h>** header. When threads executing with the scheduling policy SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC are waiting on a mutex, they acquire the mutex in priority order when the mutex is unlocked.

Application Usage

After these attributes have been set, a thread can be created with the specified attributes using **pthread_create()**. Using these routines does not affect the current running thread.

Return Values

If successful, the **pthread_attr_getschedpolicy()** and **pthread_attr_setschedpolicy()** functions return 0; otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_attr_setschedpolicy()** function might fail if:

Item	Description
EINVAL	The value of *policy* is not valid.
ENOTSUP	An attempt was made to set the attribute to an unsupported value.

These functions do not return an error code of EINTR.

pthread_attr_getstackaddr Subroutine

Purpose

Returns the value of the stackaddr attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_getstackaddr (attr, stackaddr)
const pthread_attr_t *attr;
void **stackaddr;
```

Description

The **pthread_attr_getstackaddr** subroutine returns the value of the stackaddr attribute of the thread attributes object *attr*. This attribute specifies the stack address of the thread created with this attributes object.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
attr	Specifies the thread attributes object.
stackaddr	Points to where the stackaddr attribute value will be stored.

Return Values

Upon successful completion, the value of the stackaddr attribute is returned via the *stackaddr* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_getstackaddr** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.

This function will not return EINTR.

pthread_attr_getstacksize Subroutine

Purpose

Returns the value of the stacksize attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_getstacksize (attr, stacksize)
const pthread_attr_t *attr;
size_t *stacksize;
```

Description

The **pthread_attr_getstacksize** subroutine returns the value of the stacksize attribute of the thread attributes object *attr*. This attribute specifies the minimum stacksize of a thread created with this attributes object. The value is given in bytes. For 32-bit compiled applications, the default stacksize is 96 KB (defined in the **pthread.h** file). For 64-bit compiled applications, the default stacksize is 192 KB (defined in the **pthread.h** file).

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
attr	Specifies the thread attributes object.
stacksize	Points to where the stacksize attribute value will be stored.

Return Values

Upon successful completion, the value of the stacksize attribute is returned via the *stacksize* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_getstacksize** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* or *stacksize* parameters are not valid.

This function will not return an error code of [EINTR].

pthread_attr_init Subroutine

Purpose

Creates a thread attributes object and initializes it with default values.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_init ( attr)
pthread_attr_t *attr;
```

Description

The **pthread_attr_init** subroutine creates a new thread attributes object *attr*. The new thread attributes object is initialized with the following default values:

Always initialized	
Attribute	**Default value**
Detachstate	**PTHREAD_CREATE_JOINABLE**
Contention-scope	**PTHREAD_SCOPE_SYSTEM** the default ensures compatibility with implementations that do not support this POSIX option.
Inheritsched	**PTHREAD_INHERITSCHED**
Schedparam	A **sched_param** structure which `sched_prio` field is set to 1, the least favored priority.
Schedpolicy	**SCHED_OTHER**

Always initialized *(continued)*	
Attribute	**Default value**
Stacksize	**PTHREAD_STACK_MIN**
Guardsize	**PAGESIZE**

The resulting attribute object (possibly modified by setting individual attribute values), when used by **pthread_create**, defines the attributes of the thread created. A single attributes object can be used in multiple simultaneous calls to **pthread_create**.

Parameters

Item	Description
attr	Specifies the thread attributes object to be created.

Return Values

Upon successful completion, the new thread attributes object is filled with default values and returned via the *attr* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_init** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.
ENOMEM	There is not sufficient memory to create the thread attribute object.

This function will not return an error code of [EINTR].

pthread_attr_getdetachstate or pthread_attr_setdetachstate Subroutines

Purpose

Sets and returns the value of the detachstate attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_setdetachstate (attr, detachstate)
pthread_attr_t *attr;
int detachstate;

int pthread_attr_getdetachstate (attr, detachstate)
const pthread_attr_t *attr;
int *detachstate;
```

Description

The detachstate attribute controls whether the thread is created in a detached state. If the thread is created detached, then use of the ID of the newly created thread by the **pthread_detach** or **pthread_join** function is an error.

The **pthread_attr_setdetachstate** and **pthread_attr_getdetachstate**, respectively, set and get the **detachstate** attribute in the *attr* object.

The detachstate attribute can be set to either PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE. A value of PTHREAD_CREATE_DETACHED causes all threads created with *attr* to be in the detached state, whereas using a value of PTHREAD_CREATE_JOINABLE causes all threads created with *attr* to be in the joinable state. The default value of the detachstate attribute is PTHREAD_CREATE_JOINABLE.

Parameters

Item	Description
attr	Specifies the thread attributes object.
detachstate	Points to where the detachstate attribute value will be stored.

Return Values

Upon successful completion, **pthread_attr_setdetachstate** and **pthread_attr_getdetachstate** return a value of **0**. Otherwise, an error number is returned to indicate the error.

The **pthread_attr_getdetachstate** function stores the value of the detachstate attribute in the *detachstate* parameter if successful.

Error Codes

The **pthread_attr_setdetachstate** function will fail if:

Item	Description
EINVAL	The value of *detachstate* was not valid.

The **pthread_attr_getdetachstate** and **pthread_attr_setdetachstate** functions will fail if:

Item	Description
EINVAL	The attribute parameter is invalid.

These functions will not return an error code of EINTR.

pthread_attr_getscope and pthread_attr_setscope Subroutines

Purpose

Gets and sets the scope attribute in the **attr** object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
int pthread_attr_setscope (attr, contentionscope)
pthread_attr_t *attr;
int contentionscope;

int pthread_attr_getscope (attr, contentionscope)
const pthread_attr_t *attr;
int *contentionscope;
```

Description

The scope attribute controls whether a thread is created in system or process scope.

The **pthread_attr_getscope** and **pthread_attr_setscope** subroutines get and set the scope attribute in the **attr** object.

The scope can be set to PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS. A value of PTHREAD_SCOPE_SYSTEM causes all threads created with the *attr* parameter to be in system scope, whereas a value of PTHREAD_SCOPE_PROCESS causes all threads created with the *attr* parameter to be in process scope.

The default value of the *contentionscope* parameter is PTHREAD_SCOPE_SYSTEM.

Parameters

Item	Description
attr	Specifies the thread attributes object.
contentionscope	Points to where the scope attribute value will be stored.

Return Values

Upon successful completion, the **pthread_attr_getscope** and **pthread_attr_setscope** subroutines return a value of 0. Otherwise, an error number is returned to indicate the error.

Error Codes

Item	Description
EINVAL	The value of the attribute being set/read is not valid.
ENOTSUP	An attempt was made to set the attribute to an unsupported value.

pthread_attr_getsrad_np and pthread_attr_setsrad_np Subroutines

Purpose

Gets and sets the SRAD (Scheduler Resource Allocation Domain) affinity attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_setsrad_np (attr, srad,flags)
pthread_attr_tattr;
sradid_tsrad;
```

```
int flags;
int pthread_attr_getsrad_np (attr, srad,flagsp)
pthread_attr_t *attr;
sradid_t *srad;
int *flagsp;
```

Description

The *sradp/srad* parameter specifies the SRAD that attracts a thread created with the attributes object. By default, newly created threads are balanced over the SRADs in a system in accordance with system policies.

The **pthread_attr_getsrad_np** subroutine gets the SRAD affinity attribute, while the **pthread_attr_setsrad_np** subroutine sets the SRAD affinity attribute in the thread attributes object specified by the *attr* parameter.

The *flags* parameter indicates whether the SRAD attachment is **strict** or advisory.

The *flagsp* parameter returns **R_STRICT_SRAD** if the **SRAD** attachment, if any, is **strict**.

Parameters

Item	Description
attr	Specifies the thread attributes object.
sradp	Points to a location where the **SRAD** to be extracted is stored.
srad	Specifies the **SRAD** to be extracted.
flags	Setting **R_STRICT_SRAD** indicates that the **SRAD** is a strictly preferred one. If **SRAD** attachment is NULL, set to **R_STRICT_SRAD**.
flagsp	Points to a location where the flags associated with the **SRAD** attachment, if any, is stored.

Return Values

Upon successful completion, the **pthread_attr_getsrad_np** and **pthread_attr_setsrad_np** subroutines return a value of 0. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_attr_getsrad_np** and **pthread_attr_setsrad_np** subroutines are unsuccessful if the following are true:

Item	Description
ENOTSUP	Enhanced affinity is not present or not enabled.
EINVAL (**pthread_attr_getsrad_np**)	The attribute object specified by the **attr** parameter is invalid or the address pointed by the *sradp* parameter is not aligned to hold an *sradid_t*.
EINVAL (**pthread_attr_setsrad_np**)	The SRAD affinity value specified by the *sradp* parameter is not valid.

Note: The **pthread_attr_getsrad_np**, and **pthread_attr_setsrad_np** functions do not return the error code **EINTR**.

pthread_attr_getukeyset_np or pthread_attr_setukeyset_np Subroutine

Purpose

Gets and sets the value of the active user-key-set attribute of a thread attributes object.

Library

Threads library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/ukeys.h>

int pthread_attr_getukeyset_np (attr, ukeyset)
const pthread_attr_t * attr;
ukeyset_t * ukeyset;
```

Description

The *ukeyset* parameter specifies the active user-key-set for a thread created with this attributes object. By default, newly-created threads can only access (both read and write) memory pages that have been assigned the default user-key **UKEY_PUBLIC**. User-key-sets are not inherited across the **pthread_create** subroutine.

The **pthread_attr_getukeyset_np** subroutine gets the user-key-set attribute, while the **pthread_attr_setukeyset_np** subroutine sets the user-key-set attribute in the thread attributes object specified by the *attr* parameter.

Both the **pthread_attr_getukeyset_np** and the **pthread_attr_setukeyset_np** subroutines will fail unless the **ukey_enable** subroutine has been previously successfully run by a thread in the process. Refer to the Storage Protect Keys article for more details.

Parameters

Item	Description
attr	Specifies the thread attributes object.
ukeyset	Points to a location where the user-key-set attribute value is stored.

Return Values

The **pthread_attr_getukeyset_np** and **pthread_attr_setukeyset_np** subroutines return a value of 0 on success. Otherwise, an error code is returned.

Errors Codes

The **pthread_attr_getukeyset_np** and **pthread_attr_setukeyset_np** subroutines are unsuccessful if the following are true:

Item	Description
EINVAL	The attribute object specified by the *attr* parameter is invalid or the address pointed to by the *ukeyset* parameter is not aligned to hold a user-key-set.
ENOSYS	Process is not a user-key-enabled process.

In addition, the **pthread_attr_setukeyset_np** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The user-key-set value specified by the *ukeyset* parameter is not valid.

These functions will not return an error code of **EINTR**.

pthread_attr_setschedparam Subroutine

Purpose

Sets the value of the schedparam attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>

int pthread_attr_setschedparam (attr, schedparam)
pthread_attr_t *attr;
const struct sched_param *schedparam;
```

Description

The **pthread_attr_setschedparam** subroutine sets the value of the schedparam attribute of the thread attributes object *attr*. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The `sched_priority` field of the **sched_param** structure contains the priority of the thread.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
attr	Specifies the thread attributes object.
schedparam	Points to where the scheduling parameters to set are stored. The `sched_priority` field must be in the range from 1 to 127, where 1 is the least favored priority, and 127 the most favored.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_setschedparam** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.
ENOSYS	The priority scheduling POSIX option is not implemented.
ENOTSUP	The value of the schedparam attribute is not supported.

pthread_attr_setstackaddr Subroutine

Purpose

Sets the value of the stackaddr attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_setstackaddr (attr, stackaddr)
pthread_attr_t *attr;
void *stackaddr;
```

Description

The **pthread_attr_setstackaddr** subroutine sets the value of the stackaddr attribute of the thread attributes object *attr*. This attribute specifies the stack address of a thread created with this attributes object.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

A Provision has been made in **libpthreads**to create guardpages for the user stack internally. This is used for debugging purposes only. By default, it is turned off and can be invoked by exporting the following environment variable:

```
AIXTHREAD_GUARDPAGES_FOR_USER_STACK=n (Where n is the decimal number of guard pages.)
```

Note: Even if it is exported, guard pages will only be constructed if both the stackaddr and stacksize attributes have been set by the caller for the thread. Also, the guard pages and alignment pages will be created out of the user's stack (which will reduce the stack size). If the new stack size after creating guard pages is less than the minimum stack size (PTHREAD_STACK_MIN), then the guard pages will not be constructed.

Parameters

Item	Description
attr	Specifies the thread attributes object.

Item	Description

stackaddr	Specifies the stack address to set. It is a void pointer. The address that needs to be passed is not the beginning of the malloc generated address but the beginning of the stack. For example:
	`stackaddr = malloc(stacksize);`
	`pthread_attr_setstackaddr(&thread, stackaddr + stacksize);`

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_setstackaddr** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.
ENOSYS	The stack address POSIX option is not implemented.

pthread_attr_setstacksize Subroutine

Purpose

Sets the value of the stacksize attribute of a thread attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_setstacksize (attr, stacksize)
pthread_attr_t *attr;
size_t stacksize;
```

Description

The **pthread_attr_setstacksize** subroutine sets the value of the stacksize attribute of the thread attributes object *attr*. This attribute specifies the minimum stack size, in bytes, of a thread created with this attributes object.

The allocated stack size is always a multiple of 8K bytes, greater or equal to the required minimum stack size of 56K bytes (**PTHREAD_STACK_MIN**). The following formula is used to calculate the allocated stack size: if the required stack size is lower than 56K bytes, the allocated stack size is 56K bytes; otherwise, if the required stack size belongs to the range from (56 + (*n* - 1) * 16) K bytes to (56 + *n* * 16) K bytes, the allocated stack size is (56 + *n* * 16) K bytes.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
attr	Specifies the thread attributes object.
stacksize	Specifies the minimum stack size, in bytes, to set. The default stack size is **PTHREAD_STACK_MIN**. The minimum stack size should be greater or equal than this value.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_attr_setstacksize** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid, or the value of the *stacksize* parameter exceeds a system imposed limit.
ENOSYS	The stack size POSIX option is not implemented.

pthread_attr_setsuspendstate_np and pthread_attr_getsuspendstate_np Subroutine

Purpose

Controls whether a thread is created in a suspended state.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_attr_setsuspendstate_np (attr, suspendstate)
pthread_attr_t *attr;
int suspendstate;

int pthread_attr_getsuspendstate_np (attr, suspendstate)
pthread_attr_t *attr;
int *suspendstate;
```

Description

The *suspendstate* attribute controls whether the thread is created in a suspended state. If the thread is created suspended, the thread start routine will not execute until **pthread_continue_np** is run on the thread. The **pthread_attr_setsuspendstate_np** and **pthread_attr_getsuspendstate_np** routines, respectively, set and get the *suspendstate* attribute in the *attr* object.

The *suspendstate* attribute can be set to either **PTHREAD_CREATE_SUSPENDED_NP** or **PTHREAD_CREATE_UNSUSPENDED_NP**. A value of **PTHREAD_CREATE_SUSPENDED_NP** causes all threads created with *attr* to be in the suspended state, whereas using a value of **PTHREAD_CREATE_UNSUSPENDED_NP** causes all threads created with *attr* to be in the unsuspended state. The default value of the *suspendstate* attribute is **PTHREAD_CREATE_UNSUSPENDED_NP**.

Parameters

Item	Description
attr	Specifies the thread attributes object.
suspendstate	Points to where the *suspendstate* attribute value will be stored.

Return Values

Upon successful completion, **pthread_attr_setsuspendstate_np** and **pthread_attr_getsuspendstate_np** return a value of 0. Otherwise, an error number is returned to indicate the error.

The **pthread_attr_getsuspendstate_np** function stores the value of the *suspendstate* attribute in *suspendstate* if successful.

Error Codes

The **pthread_attr_setsuspendstate_np** function will fail if:

Item	Description
EINVAL	The value of *suspendstate* is not valid.

pthread_barrier_destroy or pthread_barrier_init Subroutine

Purpose

Destroys or initializes a barrier object.

Syntax

```
#include <pthread.h>

int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_init(pthread_barrier_t *restrict barrier,
       const pthread_barrierattr_t *restrict attr, unsigned count);
```

Description

The **pthread_barrier_destroy** subroutine destroys the barrier referenced by the *barrier* parameter and releases any resources used by the barrier. The effect of subsequent use of the barrier is undefined until the barrier is reinitialized by another call to the **pthread_barrier_init** subroutine. An implementation can use this subroutine to set the *barrier* parameter to an invalid value. The results are undefined if the **pthread_barrier_destroy** subroutine is called when any thread is blocked on the barrier, or if this function is called with an uninitialized barrier.

The **pthread_barrier_init** subroutine allocates any resources required to use the barrier referenced by the *barrier* parameter and initializes the barrier with attributes referenced by the *attr* parameter. If the *attr* parameter is NULL, the default barrier attributes are used; the effect is the same as passing the address of a default barrier attributes object. The results are undefined if **pthread_barrier_init** subroutine is called when any thread is blocked on the barrier (that is, has not returned from the **pthread_barrier_wait** call). The results are undefined if a barrier is used without first being initialized. The results are undefined if the **pthread_barrier_init** subroutine is called specifying an already initialized barrier.

The *count* argument specifies the number of threads that must call the **pthread_barrier_wait** subroutine before any of them successfully return from the call. The value specified by the *count* parameter must be greater than zero.

If the **pthread_barrier_init** subroutine fails, the barrier is not initialized and the contents of barrier are undefined.

Only the object referenced by the *barrier* parameter can be used for performing synchronization. The result of referring to copies of that object in calls to the **pthread_barrier_destroy** or **pthread_barrier_wait** subroutine is undefined.

Return Values

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

Error Codes

The **pthread_barrier_destroy** subroutine can fail if:

Item	Description
EBUSY	The implementation has detected an attempt to destroy a barrier while it is in use (for example, while being used in a **pthread_barrier_wait** call) by another thread.
EINVAL	The value specified by barrier is invalid.

The pthread_barrier_init() function will fail if:

Item	Description
EAGAIN	The system lacks the necessary resources to initialize another barrier.
EINVAL	The value specified by the *count* parameter is equal to zero.
ENOMEM	Insufficient memory exists to initialize the barrier.

The **pthread_barrier_init** subroutine can fail if:

Item	Description
EBUSY	The implementation has detected an attempt to reinitialize a barrier while it is in use (for example, while being used in a **pthread_barrier_wait** call) by another thread.
EINVAL	The value specified by the *attr* parameter is invalid.

pthread_barrier_wait Subroutine

Purpose

Synchronizes threads at a barrier.

Syntax

```
#include <pthread.h>

int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Description

The **pthread_barrier_wait** subroutine synchronizes participating threads at the barrier referenced by barrier. The calling thread blocks until the required number of threads have called **pthread_barrier_wait**specifying the barrier.

When the required number of threads have called **pthread_barrier_wait**specifying the barrier, the constant **PTHREAD_BARRIER_SERIAL_THREAD** is returned to one unspecified thread and 0 is returned to the remaining threads. At this point, the barrier resets to the state it had as a result of the most recent **pthread_barrier_init** function that referenced it.

The constant **PTHREAD_BARRIER_SERIAL_THREAD** is defined in **<pthread.h>**, and its value is distinct from any other value returned by **pthread_barrier_wait**.

The results are undefined if this function is called with an uninitialized barrier.

If a signal is delivered to a thread blocked on a barrier, upon return from the signal handler, the thread resumes waiting at the barrier if the barrier wait has not completed (that is, if the required number of threads have not arrived at the barrier during the execution of the signal handler); otherwise, the thread continues as normal from the completed barrier wait. Until the thread in the signal handler returns from it, other threads might proceed past the barrier after they have all reached it.

Note: When the required number of threads has called **pthread_barrier_wait**, the **PTHREAD_BARRIER_SERIAL_THREAD** constant is returned by the last pthread that called **pthread_barrier_wait**. Furthermore, if a thread is in a signal handler while waiting and all the required threads have reached the barrier, the other threads can proceed past the barrier.

A thread that has blocked on a barrier does not prevent any unblocked thread that is eligible to use the same processing resources from eventually making forward progress in its execution. Eligibility for processing resources is determined by the scheduling policy.

Parameters

Item	Description
barrier	Points to the barrier where participating threads wait.

Return Values

Upon successful completion, **pthread_barrier_wait** returns **PTHREAD_BARRIER_SERIAL_THREAD** for a single (arbitrary) thread synchronized at the barrier and 0 for the other threads. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_barrier_destroy** subroutine can fail if:

Item	Description
EINVAL	The value specified by *barrier* does not refer to an initialized barrier object.

This function does not return an error code of **EINTR**.

pthread_barrierattr_destroy or pthread_barrierattr_init Subroutine

Purpose

Destroys or initializes the barrier attributes object.

Syntax

```
#include <pthread.h>

int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

Description

The **pthread_barrierattr_destroy** subroutine destroys a barrier attributes object. A destroyed *attr* attributes object can be reinitialized using the **pthread_barrierattr_init** subroutine; the results of otherwise referencing the object after it has been destroyed are undefined. An implementation can cause

the **pthread_barrierattr_destroy** subroutine to set the object referenced by the *attr* parameter to an invalid value.

The **pthread_barrierattr_init** subroutine initializes a barrier attributes object *attr* with the default value for all of the attributes defined by the implementation.

Results are undefined if the **pthread_barrierattr_init** subroutine is called specifying an already initialized *attr* attributes object.

After a barrier attributes object has been used to initialize one or more barriers, any function affecting the attributes object (including destruction) do not affect any previously initialized barrier.

Return Values

If successful, the **pthread_barrierattr_destroy** and **pthread_barrierattr_init** subroutines return zero; otherwise, an error number shall be returned to indicate the error.

Error Codes

The **pthread_barrierattr_destroy** subroutine can fail if:

Item	Description
EINVAL	The value specified by the *attr* parameter is invalid.

The **pthread_barrierattr_init** subroutine will fail if:

Item	Description
ENOMEM	Insufficient memory exists to initialize the barrier attributes object.

pthread_barrierattr_getpshared or pthread_barrierattr_setpshared Subroutine

Purpose

Gets and sets the process-shared attribute of the barrier attributes object.

Syntax

```
#include <pthread.h>

int pthread_barrierattr_getpshared(const pthread_barrierattr_t *
       restrict attr, int *restrict pshared);
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
       int pshared);
```

Description

The **pthread_barrierattr_getpshared** subroutine obtains the value of the process-shared attribute from the attributes object referenced by the *attr* parameter. The **pthread_barrierattr_setpshared** subroutine sets the process-shared attribute in an initialized attributes object referenced by the *attr* parameter.

The process-shared attribute is set to **PTHREAD_PROCESS_SHARED** to permit a barrier to be operated upon by any thread that has access to the memory where the barrier is allocated. If the process-shared attribute is **PTHREAD_PROCESS_PRIVATE**, the barrier is only operated upon by threads created within the same process as the thread that initialized the barrier; if threads of different processes attempt to operate on such a barrier, the behavior is undefined. The default value of the attribute is **PTHREAD_PROCESS_PRIVATE**. Both constants **PTHREAD_PROCESS_SHARED** and **PTHREAD_PROCESS_PRIVATE** are defined in the **pthread.h** file.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation-defined.

Return Values

If successful, the **pthread_barrierattr_getpshared** subroutine will return zero and store the value of the process-shared attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number shall be returned to indicate the error.

If successful, the **pthread_barrierattr_setpshared** subroutine will return zero; otherwise, an error number shall be returned to indicate the error.

Error Codes

These functions may fail if:

Item	Description
EINVAL	The value specified by *attr* is invalid.

The **pthread_barrierattr_setpshared** subroutine will fail if:

Item	Description
EINVAL	The new value specified for the process-shared attribute is not one of the legal values **PTHREAD_PROCESS_SHARED** or **PTHREAD_PROCESS_PRIVATE**.

pthread_cancel Subroutine

Purpose

Requests the cancellation of a thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_cancel (thread)
pthread_t thread;
```

Description

The **pthread_cancel** subroutine requests the cancellation of the thread *thread*. The action depends on the cancelability of the target thread:

- If its cancelability is disabled, the cancellation request is set pending.
- If its cancelability is deferred, the cancellation request is set pending till the thread reaches a cancellation point.
- If its cancelability is asynchronous, the cancellation request is acted upon immediately; in some cases, it may result in unexpected behavior.

The cancellation of a thread terminates it safely, using the same termination procedure as the **pthread_exit** subroutine.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
thread	Specifies the thread to be canceled.

Return Values

If successful, the **pthread_cancel** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **ptread_cancel** function may fail if:

Item	Description
ESRCH	No thread could be found corresponding to that specified by the given thread ID.

The **pthread_cancel** function will not return an error code of EINTR.

pthread_cleanup_pop or pthread_cleanup_push Subroutine

Purpose

Activates and deactivates thread cancellation handlers.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

void pthread_cleanup_pop (execute)
int execute;

void pthread_cleanup_push (routine, arg)
void (*routine)(void *);
void *arg;
```

Description

The **pthread_cleanup_push** subroutine pushes the specified cancellation cleanup handler *routine* onto the calling thread's cancellation cleanup stack. The cancellation cleanup handler is popped from the cancellation cleanup stack and invoked with the argument *arg* when: (a) the thread exits (that is, calls **pthread_exit**, (b) the thread acts upon a cancellation request, or (c) the thread calls **pthread_cleanup_pop** with a nonzero *execute* argument.

The **pthread_cleanup_pop** subroutine removes the subroutine at the top of the calling thread's cancellation cleanup stack and optionally invokes it (if *execute* is nonzero).

These subroutines may be implemented as macros and will appear as statements and in pairs within the same lexical scope (that is, the **pthread_cleanup_push** macro may be thought to expand to a token

list whose first token is **'{'** with **pthread_cleanup_pop** expanding to a token list whose last token is the corresponding **'}'**).

The effect of calling **longjmp** or **siglongjmp** is undefined if there have been any calls to **pthread_cleanup_push** or **pthread_cleanup_pop** made without the matching call since the jump buffer was filled. The effect of calling **longjmp** or **siglongjmp** from inside a cancellation cleanup handler is also undefined unless the jump buffer was also filled in the cancellation cleanup handler.

Parameters

Item	Description
execute	Specifies if the popped subroutine will be executed.
routine	Specifies the address of the cancellation subroutine.
arg	Specifies the argument passed to the cancellation subroutine.

pthread_cond_destroy or pthread_cond_init Subroutine

Purpose

Initialize and destroys condition variables.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_cond_init (cond, attr)
pthread_cond_t *cond;
const pthread_condattr_t *attr;

int pthread_cond_destroy (cond)
pthread_cond_t *cond;

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Description

The function **pthread_cond_init** initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used; the effect is the same as passing the address of a default condition variable attributes object. Upon successful initialization, the state of the condition variable becomes initialized.

Attempting to initialize an already initialized condition variable results in undefined behavior.

The function **pthread_cond_destroy** destroys the given condition variable specified by *cond*; the object becomes, in effect, uninitialized. An implementation may cause **pthread_cond_destroy** to set the object referenced by *cond* to an invalid value. A destroyed condition variable object can be re-initialized using **pthread_cond_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are currently blocked results in undefined behavior.

In cases where default condition variable attributes are appropriate, the macro PTHREAD_COND_INITIALIZER can be used to initialize condition variables that are statically allocated.

The effect is equivalent to dynamic initialization by a call to **pthread_cond_init** with parameter *attr* specified as NULL, except that no error checks are performed.

Parameters

Item	Description
cond	Pointer to the condition variable.
attr	Specifies the attributes of the condition.

Return Values

If successful, the **pthread_cond_init** and **pthread_cond_destroy** functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks, if implemented, act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the condition variable specified by *cond*.

Error Codes

The **pthread_cond_init** function will fail if:

Item	Description
EAGAIN	The system lacked the necessary resources (other than memory) to initialize another condition variable.
ENOMEM	Insufficient memory exists to initialize the condition variable.

The **pthread_cond_init** function may fail if:

Item	Description
EINVAL	The value specified by *attr* is invalid.

The **pthread_cond_destroy** function may fail if:

Item	Description
EBUSY	The implementation has detected an attempt to destroy the object referenced by *cond* while it is referenced (for example, while being used in a **pthread_cond_wait** or **pthread_cond_timedwait** by another thread.
EINVAL	The value specified by *cond* is invalid.

These functions will not return an error code of EINTR.

PTHREAD_COND_INITIALIZER Macro

Purpose

Initializes a static condition variable with default attributes.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Description

The **PTHREAD_COND_INITIALIZER** macro initializes the static condition variable *cond*, setting its attributes to default values. This macro should only be used for static condition variables, since no error checking is performed.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

pthread_cond_signal or pthread_cond_broadcast Subroutine

Purpose

Unblocks one or more threads blocked on a condition.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
int pthread_cond_signal (condition)
pthread_cond_t *condition;
```

```
int pthread_cond_broadcast (condition)
pthread_cond_t *condition;
```

Description

These subroutines unblock one or more threads blocked on the condition specified by *condition*. The **pthread_cond_signal** subroutine unblocks at least one blocked thread, while the **pthread_cond_broadcast** subroutine unblocks all the blocked threads.

If more than one thread is blocked on a condition variable, the scheduling policy determines the order in which threads are unblocked. When each thread unblocked as a result of a **pthread_cond_signal** or **pthread_cond_broadcast** returns from its call to **pthread_cond_wait** or **pthread_cond_timedwait**, the thread owns the mutex with which it called **pthread_cond_wait**or **pthread_cond_timedwait**. The thread(s) that are unblocked contend for the mutex according to the scheduling policy (if applicable), and as if each had called **pthread_mutex_lock**.

The **pthread_cond_signal** or **pthread_cond_broadcast** functions may be called by a thread whether or not it currently owns the mutex that threads calling **pthread_cond_wait** or **pthread_cond_timedwait** have associated with the condition variable during their waits; however, if predictable scheduling behavior is required, then that mutex is locked by the thread calling **pthread_cond_signal** or **pthread_cond_broadcast**.

If no thread is blocked on the condition, the subroutine succeeds, but the signalling of the condition is not held. The next thread calling **pthread_cond_wait** will be blocked.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
condition	Specifies the condition to signal.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Code

The **pthread_cond_signal** and **pthread_cond_broadcast** subroutines are unsuccessful if the following is true:

Item	Description
EINVAL	The *condition* parameter is not valid.

pthread_cond_wait or pthread_cond_timedwait Subroutine

Purpose

Blocks the calling thread on a condition.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_cond_wait (cond, mutex)
pthread_cond_t *cond;
pthread_mutex_t *mutex;

int pthread_cond_timedwait (cond, mutex, timeout)
pthread_cond_t *cond;
pthread_mutex_t *mutex;
const struct timespec *timeout;
```

Description

The **pthread_cond_wait** and **pthread_cond_timedwait** functions are used to block on a condition variable. They are called with *mutex* locked by the calling thread or undefined behavior will result.

These functions atomically release *mutex* and cause the calling thread to block on the condition variable *cond*; atomically here means atomically with respect to access by another thread to the mutex and then the condition variable. That is, if another thread is able to acquire the mutex after the about-to-block thread releases it, then a subsequent call to **pthread_cond_signal** or **pthread_cond_broadcast** in that thread behaves as if it were issued after the about-to-block thread has blocked.

Upon successful return, the mutex is locked and owned by the calling thread.

When you use condition variables, there is always a Boolean predicate involving shared variable that is associated with each condition wait that is true if the thread must proceed. Spurious wakeups from the **pthread_cond_wait** or **pthread_cond_timedwait** functions might occur. Since the return from **pthread_cond_wait** or **pthread_cond_timedwait** does not imply anything about the value of this predicate, the predicate must be reevaluated upon such return.

The effect of using more than one mutex for concurrent **pthread_cond_wait** or **pthread_cond_timedwait** operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when the wait returns.

A condition wait (whether timed or not) is a cancellation point. When the cancelability enable state of a thread is set to PTHREAD_CANCEL_DEFERRED, a side effect of acting upon a cancellation request while in a condition wait is that the mutex is (in effect) reacquired before calling the first cancellation cleanup handler. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the call to **pthread_cond_wait** or **pthread_cond_timedwait**, but at that point notices the cancellation request and instead of returning to the caller of **pthread_cond_wait** or **pthread_cond_timedwait**, starts the thread cancellation activities, which include calling cancellation cleanup handlers.

A thread that is unblocked because it is canceled while blocked in a call to **pthread_cond_wait** or **pthread_cond_timedwait** does not consume any condition signal that may be directed concurrently at the condition variable if there are other threads blocked on the condition variable.

The **pthread_cond_timedwait** function is the same as **pthread_cond_wait** except that an error is returned if the absolute time specified by *timeout* passes (that is, system time equals or exceeds *timeout*) before the condition *cond* is signaled or broadcast, or if the absolute time that is specified by *timeout* has already been passed at the time of the call. When such time-outs occur, **pthread_cond_timedwait** releases the mutex and reacquires the mutex referenced by *mutex*. The function **pthread_cond_timedwait** is also a cancellation point. The absolute time that is specified by *timeout* can be either based on the system realtime clock or the system monotonic clock. The reference clock for the condition variable is set by calling **pthread_condattr_setclock** before its initialization with the corresponding condition attributes object.

If a signal is delivered to a thread they is waiting for a condition variable, upon return from the signal handler the thread resumes waiting for the condition variable as if it was not interrupted, or it returns zero due to spurious wakeup.

Parameters

Item	Description
cond	Specifies the condition variable to wait on.
mutex	Specifies the mutex that is used to protect the condition variable. The mutex must be locked when the subroutine is called.
timeout	Points to the absolute time structure that is specifying the blocked state timeout.

Return Values

Except if ETIMEDOUT, all these error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return, in effect, before modifying the state of the mutex specified by mutex or the condition variable specified by *cond*.

Upon successful completion, a value of zero is returned. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_cond_timedwait** function fails if:

Item	Description
ETIMEDOUT	The time specified by *timeout* to **pthread_cond_timedwait** has passed.

The **pthread_cond_wait** and **pthread_cond_timedwait** subroutines fail if the following error codes are returned:

Item	Description
EINVAL	The value specified by *cond*, *mutex*, or *timeout* is invalid.
EINVAL	Different mutexes were supplied for concurrent **pthread_cond_wait** or **pthread_cond_timedwait** operations on the same condition variable.
EINVAL	The mutex was not owned by the current thread at the time of the call.
EPERM	The mutex was not owned by the current thread at the time of the call, XPG_SUS_ENV is set to ON, and XPG_UNIX98 is not set.
ENOTRECOVERABLE	The protected state of the mutex cannot be recovered.
EOWNERDEAD	The mutex is a robust mutex, and the process of the thread that owns the mutex terminated while holding the mutex lock.

These subroutines do not return an **EINTR** error code.

pthread_condattr_destroy or pthread_condattr_init Subroutine

Purpose

Initializes and destroys condition variable.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_condattr_destroy (attr)
pthread_condattr_t *attr;

int pthread_condattr_init (attr)
pthread_condattr_t *attr;
```

Description

The function **pthread_condattr_init** initializes a condition variable attributes object *attr* with the default value for all of the attributes defined by the implementation. Attempting to initialize an already initialized condition variable attributes object results in undefined behavior.

After a condition variable attributes object has been used to initialize one or more condition variables, any function affecting the attributes object (including destruction) does not affect any previously initialized condition variables.

The **pthread_condattr_destroy** function destroys a condition variable attributes object; the object becomes, in effect, uninitialized. The **pthread_condattr_destroy** subroutine may set the object referenced by *attr* to an invalid value. A destroyed condition variable attributes object can be re-initialized using **pthread_condattr_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

Parameter

Item	Description
attr	Specifies the condition attributes object to delete.

Return Values

If successful, the **pthread_condattr_init** and **pthread_condattr_destroy** functions return zero. Otherwise, an error number is returned to indicate the error.

Error Code

The **pthread_condattr_init** function will fail if:

Item	Description
ENOMEM	Insufficient memory exists to initialize the condition variable attributes object.

The **pthread_condattr_destroy** function may fail if:

Item	Description
EINVAL	The value specified by *attr* is invalid.

These functions will not return an error code of EINTR.

pthread_condattr_getclock, pthread_condattr_setclock Subroutine

Purpose

Gets and sets the clock selection condition variable attribute.

Syntax

```
 int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
       clockid_t *restrict clock_id);
int pthread_condattr_setclock(pthread_condattr_t *attr,
       clockid_t clock_id);
```

Description

The **pthread_condattr_getclock** subroutine obtains the value of the clock attribute from the attributes object referenced by the *attr* argument. The **pthread_condattr_setclock** subroutine sets the clock attribute in an initialized attributes object referenced by the *attr* argument. If **pthread_condattr_setclock** is called with a *clock_id* argument that refers to a CPU-time clock, the call will fail.

The clock attribute is the clock ID of the clock that shall be used to measure the timeout service of the **pthread_cond_timedwait** subroutine. The default value of the clock attribute refers to the system clock.

Parameters

Item	Description
attr	Specifies the condition attributes object.
clock_id	For **pthread_condattr_getclock()**, points to where the clock attribute value will be stored.For **pthread_condattr_setclock()**, specifies the clock to set. Valid values are: **CLOCK_REALTIME** The system realtime clock. **CLOCK_MONOTONIC** The system monotonic clock. The value of this clock represents the amount of time since an unspecified point in the past. The value of this clock always grows: it cannot be set by **clock_settime()** and cannot have backward clock jumps.

Return Values

If successful, the **pthread_condattr_getclock** subroutine returns 0 and stores the value of the clock attribute of *attr* in the object referenced by the *clock_id* argument. Otherwise, an error code is returned to indicate the error.

If successful, the **pthread_condattr_setclock** subroutine returns 0; otherwise, an error code is returned to indicate the error.

Error Codes

Item	Description
EINVAL	The value specified by *attr* is invalid.
EINVAL	The **pthread_condattr_setclock** subroutine returns this error if the value specified by the *clock_id* does not refer to a known clock, or is a CPU-time clock.
ENOTSUP	The function is not supported with checkpoint-restart processes.

pthread_condattr_getpshared Subroutine

Purpose

Returns the value of the pshared attribute of a condition attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_condattr_getpshared (attr, pshared)
const pthread_condattr_t *attr;
int *pshared;
```

Description

The **pthread_condattr_getpshared** subroutine returns the value of the pshared attribute of the condition attribute object *attr*. This attribute specifies the process sharing of the condition variable created with this attributes object. It may have one of the following values:

Item	Description
PTHREAD_PROCESS_SHARED	Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes.
PTHREAD_PROCESS_PRIVATE	Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default value.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
attr	Specifies the condition attributes object.
pshared	Points to where the pshared attribute value will be stored.

Return Values

Upon successful completion, the value of the pshared attribute is returned via the *pshared* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_condattr_getpshared** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.
ENOSYS	The process sharing POSIX option is not implemented.

pthread_condattr_setpshared Subroutine

Purpose

Sets the value of the pshared attribute of a condition attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_condattr_setpshared (attr, pshared)
pthread_condattr_t *attr;
int pshared;
```

Description

The **pthread_condattr_setpshared** subroutine sets the value of the pshared attribute of the condition attributes object *attr*. This attribute specifies the process sharing of the condition variable created with this attributes object.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
attr	Specifies the condition attributes object.

Item	Description
pshared	Specifies the process sharing to set. It must have one of the following values: **PTHREAD_PROCESS_SHARED** Specifies that the condition variable can be used by any thread that has access to the memory where it is allocated, even if these threads belong to different processes. **PTHREAD_PROCESS_PRIVATE** Specifies that the condition variable shall only be used by threads within the same process as the thread that created it. This is the default value.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_condattr_setpshared** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* or *pshared* parameters are not valid.

pthread_create Subroutine

Purpose

Creates a new thread, initializes its attributes, and makes it runnable.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_create (thread, attr, start_routine (void *), arg)
pthread_t *thread;
const pthread_attr_t *attr;
void *(*start_routine) (void *);
void *arg;
```

Description

The **pthread_create** subroutine creates a new thread and initializes its attributes using the thread attributes object specified by the *attr* parameter. The new thread inherits its creating thread's signal mask; but any pending signal of the creating thread will be cleared for the new thread.

The new thread is made runnable, and will start executing the *start_routine* routine, with the parameter specified by the *arg* parameter. The *arg* parameter is a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (**int** for example), because the casts may not be portable.

After thread creation, the thread attributes object can be reused to create another thread, or deleted.

The thread terminates in the following cases:

- The thread returned from its starting routine (the **main** routine for the initial thread)
- The thread called the **pthread_exit** subroutine

- The thread was canceled
- The thread received a signal that terminated it
- The entire process is terminated due to a call to either the **exec** or **exit** subroutines.

  Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

When multiple threads are created in a process, the **FULL_CORE** flag is set for all signals. This means that if a core file is produced, it will be much bigger than a single_threaded application. This is necessary to debug multiple-threaded processes.

When a process uses the **pthread_create** function, and thus becomes multi-threaded, the **FULL_CORE** flag is enabled for all signals. If a signal is received whose action is to terminate the process with a core dump, a full dump (usually much larger than a regular dump) will be produced. This is necessary so that multi-threaded programs can be debugged with the **dbx** command.

The following piece of pseudocode is an example of how to avoid getting a full core. Please note that in this case, debug will not be possible. It may be easier to limit the size of the core with the **ulimit** command.

```
struct sigaction siga;
siga.sa_handler = SIG_DFL;
siga.sa_flags = SA_RESTART;
SIGINITSET(siga.as_mask);
sigaction(<SIGNAL_NUMBER>, &siga, NULL);
```

The alternate stack is not inherited.

Parameters

Item	Description
thread	Points to where the thread ID will be stored.
attr	Specifies the thread attributes object to use in creating the thread. If the value is **NULL**, the default attributes values will be used.
start_routine	Points to the routine to be executed by the thread.
arg	Points to the single argument to be passed to the *start_routine* routine.

Return Values

If successful, the **pthread_create** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_create** function will fail if:

Item	Description
EAGAIN	If WLM is running, the limit on the number of threads in the class is reached.
EAGAIN	The limit on the number of threads per process has been reached.
EINVAL	The value specified by **attr** is not valid.
EPERM	The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

The **pthread_create** function will not return an error code of **EINTR**.

pthread_create_withcred_np Subroutine

Purpose

Creates a new thread with a new set of credentials, initializes its attributes, and makes it runnable.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/cred.h>

int pthread_create_withcred_np(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine)(void),
void *arg, struct __pthrdscreds *credp)
```

Description

The **pthread_create_withcred_np** subroutine is equivalent to the **pthread_create** routine except that it allows the new thread to be created and start running with the credentials specified by the *credp* parameter. Only a process that has the credentials capability or is running with an effective user ID as the root user is allowed to modify its credentials using this routine.

You can modify the following credentials:

- Effective, real and saved user IDs
- Effective, real and saved group IDs
- Supplementary group IDs

Note: The administrator can set the lowest user ID value to which a process with credentials capability is allowed to switch its user IDs. A value of 0 can be specified for any of the preceding credentials to indicate that the thread should inherit that specific credential from its caller. The administrator can also set the lowest group ID to which a process with credentials capability is allowed to switch its group IDs.

The *__pc_flags* flag field in the *credp* parameter provides options to inherit credentials from the parent thread.

The newly created thread runs with per-thread credentials, and system calls such as **getuid** or **getgid** returns the thread's credentials. Similarly, when a file is opened or a message is received, the thread's credentials are used to determine whether the thread has the privilege to execute the operation.

Parameters

Item	Description
thread	Points to the location where the thread ID is stored.
attr	Specifies the thread attributes object to use while creating the thread. If the value is NULL, the default attributes values are used.
start_routine	Points to the routine to be executed by the thread.
arg	Points to the single argument to be passed to the **start_routine** routine.

Item	Description
credp	Points to a structure of type **__pthrdscreds**, that contains the credentials structure and the inheritance flags. If set to NULL, the **pthread_create_withcred_np** subroutine is the same as the **pthread_create** routine. The *__pc_cred* field indicates the credentials to be assigned to the new pthread.The *__pc_flags* field indicates which credentials, if any, are to be inherited from the parent thread. This field is constructed by logically OR'ing one or more of the following values: **PTHRDSCREDS_INHERIT_UIDS** Inherit user IDs from the parent thread. **PTHRDSCREDS_INHERIT_GIDS** Inherit group IDs from the parent thread. **PTHRDSCREDS_INHERIT_GSETS** Inherit the group sets from the parent thread. **PTHRDSCREDS_INHERIT_CAPS** Inherit capabilities from the parent thread. **PTHRDSCREDS_INHERIT_PRIVS** Inherit privileges from the parent thread. **PTHRDSCREDS_INHERIT_ALL** Inherit all the credentials from the parent thread.

Security

Only a process that has the credentials capability or is running with an effective user ID (such as the root user) is allowed to modify its credentials using this routine.

Return Values

If successful, the **pthread_create_withcred_np** subroutine returns 0. Otherwise, an error number is returned to indicate the error.

Error Codes

Item	Description
EAGAIN	If WLM is running, the limit on the number of threads in the class might have been met.
EFAULT	The *credp* parameter points to a location outside of the allocated address space of the process.
EINVAL	The credentials specified in the *credp* parameter are not valid.
EPERM	The caller does not have appropriate permission to set the credentials.

The **pthread_create_withcred_np** subroutine does not return an error code of **EINTR**.

pthread_delay_np Subroutine

Purpose

Causes a thread to wait for a specified period.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_delay_np ( interval)
struct timespec *interval;
```

Description

The **pthread_delay_np** subroutine causes the calling thread to delay execution for a specified period of elapsed wall clock time. The period of time the thread waits is at least as long as the number of seconds and nanoseconds specified in the *interval* parameter.

Note:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_delay_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

Item	Description
interval	Points to the time structure specifying the wait period.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_delay_np** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *interval* parameter is not valid.

pthread_equal Subroutine

Purpose

Compares two thread IDs.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_equal (thread1, thread2)
pthread_t thread1;
pthread_t thread2;
```

Description

The **pthread_equal** subroutine compares the thread IDs *thread1* and *thread2*. Since the thread IDs are opaque objects, it should not be assumed that they can be compared using the equality operator (==).

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
thread1	Specifies the first ID to be compared.
thread2	Specifies the second ID to be compared.

Return Values

The **pthread_equal** function returns a nonzero value if *thread1* and *thread2* are equal; otherwise, zero is returned.

If either *thread1* or *thread2* are not valid thread IDs, the behavior is undefined.

pthread_exit Subroutine

Purpose

Terminates the calling thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

void pthread_exit (status)
void *status;
```

Description

The **pthread_exit** subroutine terminates the calling thread safely, and stores a termination status for any thread that may join the calling thread. The termination status is always a void pointer; it can reference any kind of data. It is not recommended to cast this pointer into a scalar data type (**int** for example), because the casts may not be portable. This subroutine never returns.

Unlike the **exit** subroutine, the **pthread_exit** subroutine does not close files. Thus any file opened and used only by the calling thread must be closed before calling this subroutine. It is also important to note that the **pthread_exit** subroutine frees any thread-specific data, including the thread's stack. Any data allocated on the stack becomes invalid, since the stack is freed and the corresponding memory may be reused by another thread. Therefore, thread synchronization objects (mutexes and condition variables) allocated on a thread's stack must be destroyed before the thread calls the **pthread_exit** subroutine.

Returning from the initial routine of a thread implicitly calls the **pthread_exit** subroutine, using the return value as parameter.

If the thread is not detached, its resources, including the thread ID, the termination status, the thread-specific data, and its storage, are all maintained until the thread is detached or the process terminates.

If another thread joins the calling thread, that thread wakes up immediately, and the calling thread is automatically detached.

If the thread is detached, the cleanup routines are popped from their stack and executed. Then the destructor routines from the thread-specific data are executed. Finally, the storage of the thread is reclaimed and its ID is freed for reuse.

Terminating the initial thread by calling this subroutine does not terminate the process, it just terminates the initial thread. However, if all the threads in the process are terminated, the process is terminated by implicitly calling the **exit** subroutine with a return code of 0 if the last thread is detached, or 1 otherwise.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
status	Points to an optional termination status, used by joining threads. If no termination status is desired, its value should be **NULL**.

Return Values

The **pthread_exit** function cannot return to its caller.

Errors

No errors are defined.

The **pthread_exit** function will not return an error code of EINTR.

pthread_get_expiration_np Subroutine

Purpose

Obtains a value representing a desired expiration time.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
int pthread_get_expiration_np ( delta,  abstime)
struct timespec *delta;
struct timespec *abstime;
```

Description

The **pthread_get_expiration_np** subroutine adds the interval *delta* to the current absolute system time and returns a new absolute time. This new absolute time can be used as the expiration time in a call to the **pthread_cond_timedwait** subroutine.

Note:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_get_expiration_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

Item	Description
delta	Points to the time structure specifying the interval.
abstime	Points to where the new absolute time will be stored.

Return Values

Upon successful completion, the new absolute time is returned via the *abstime* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_get_expiration_np** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *delta* or *abstime* parameters are not valid.

pthread_getconcurrency or pthread_setconcurrency Subroutine

Purpose

Gets or sets level of concurrency.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_getconcurrency (void);

int pthread_setconcurrency (new_level)
int new_level;
```

Description

The **pthread_setconcurrency** subroutine allows an application to inform the threads implementation of its desired concurrency level, *new_level*. The actual level of concurrency provided by the implementation as a result of this function call is unspecified.

If *new_level* is zero, it causes the implementation to maintain the concurrency level at its discretion as if **pthread_setconcurrency** was never called.

The **pthread_getconcurrency** subroutine returns the value set by a previous call to the **pthread_setconcurrency** subroutine. If the **pthread_setconcurrency** subroutine was not previously called, this function returns zero to indicate that the implementation is maintaining the concurrency level.

When an application calls **pthread_setconcurrency**, it is informing the implementation of its desired concurrency level. The implementation uses this as a hint, not a requirement.

Use of these subroutines changes the state of the underlying concurrency upon which the application depends. Library developers are advised to not use the **pthread_getconcurrency** and **pthread_setconcurrency** subroutines since their use may conflict with an applications use of these functions.

Parameters

Item	Description
new_level	Specifies the value of the concurrency level.

Return Value

If successful, the **pthread_setconcurrency** subroutine returns zero. Otherwise, an error number is returned to indicate the error.

The **pthread_getconcurrency** subroutine always returns the concurrency level set by a previous call to **pthread_setconcurrency**. If the **pthread_setconcurrency** subroutine has never been called, **pthread_getconcurrency** returns zero.

Error Codes

The **pthread_setconcurrency** subroutine will fail if:

Item	Description
EINVAL	The value specified by *new_level* is negative.
EAGAIN	The value specific by *new_level* would cause a system resource to be exceeded.

pthread_getcpuclockid Subroutine

Purpose

Accesses a thread CPU-time clock.

Syntax

```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock_id);
```

Description

The **pthread_getcpuclockid** subroutine returns in the *clock_id* parameter the clock ID of the CPU-time clock of the thread specified by *thread_id*, if the thread specified by *thread_id* exists.

Parameters

Item	Description
thread_id	Specifies the ID of the pthread whose clock ID is requested.
clock_id	Points to the **clockid_t** structure used to return the thread CPU-time clock ID of *thread_id*.

Return Values

Upon successful completion, the **pthread_getcpuclockid** subroutine returns 0; otherwise, an error number is returned to indicate the error.

Error Codes

Item	Description
ENOTSUP	The subroutine is not supported with checkpoint-restart'ed processes.
ESRCH	The value specified by *thread_id* does not refer to an existing thread.

pthread_getiopri_np or pthread_setiopri_np Subroutine

Purpose

Sets and gets the I/O priority of a specified pthread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/extendio.h>

int pthread_getiopri_np( pthread,  *pri)
int pthread_setiopri_np( pthread,  pri)
pthread_t pthread;
iopri_t pri;
```

Description

The **pthread_getiopri_np** subroutine stores the I/O scheduling priority of the pthread into the *pri* argument. The **pthread_setiopri_np** subroutine sets the I/O scheduling priority to the *pri* argument of the specified pthread.

AIX provides the ability to prioritize I/O buffers on a per-I/O and per-process basis. With the **pthread_getiopri_np** subroutine and the **pthread_setiopri_np** subroutine, AIX provides the ability to prioritize I/O buffers on a per-thread basis.

Note: Both subroutines are only supported in a System Scope (1:1) environment.

Parameters

Item	Description
pthread	Specifies the target thread.
pri	I/O priority field used to set or store the current I/O priority of the pthread.

Return Values

Upon successful completion, the **pthread_getiopri_np** subroutine or the **pthread_setiopri_np** subroutine returns zero. A non-zero value indicates an error.

Error Codes

If any of the following conditions occur, the **pthread_getiopri_np** subroutine and the **pthread_setiopri_np** subroutine fail and return the corresponding value:

Item	Description
ESRCH	The provided pthread is not valid.
ENOTSUP	This function was called in a Process Scope (M:N) environment.
EPERM	The caller does not have the valid Role Based Access Control (RBAC) permissions (the ACT_P_GETPRI permission for the **pthread_getiopri_np** subroutine, the ACT_P_SETPRI permission for the **pthread_setiopri_np** subroutine).
EINVAL	The specified I/O priority is not valid.

pthread_getrusage_np Subroutine

Purpose

Enable or disable pthread library resource collection, and retrieve resource information for any pthread in the current process.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_getrusage_np (Ptid, RUsage, Mode)
pthread_t Ptid;
struct rusage *RUsage;
int Mode;
```

Description

The **pthread_getrusage_np** subroutine enables and disables resource collection in the pthread library and collects resource information for any pthread in the current process. When compiled in 64-bit mode, resource usage (rusage) counters are 64-bits for the calling thread. When compiled in 32-bit mode, rusage counters are 32-bits for the calling pthread.

This functionality is enabled by default. The previous **AIXTHREAD_ENRUSG** used with **pthread_getrusage_np** is no longer supported.

Parameters

Item	Description
Ptid	Specifies the target thread. Must be within the current process.

Item	Description
RUsage	Points to a buffer described in the **/usr/include/sys/resource.h** file. The fields are defined as follows: **ru_utime** The total amount of time running in user mode. **ru_stime** The total amount of time spent in the system executing on behalf of the processes. **ru_maxrss** The maximum size, in kilobytes, of the used resident set size. **ru_ixrss** An integral value indicating the amount of memory used by the text segment that was also shared among other processes. This value is expressed in *units of kilobytes* X *seconds-of-execution* and is calculated by adding the number of shared memory pages in use each time the internal system clock ticks, and then averaging over one-second intervals. **ru_idrss** An integral value of the amount of unshared memory in the data segment of a process, which is expressed in *units of kilobytes* X *seconds-of-execution*. **ru_minflt** The number of page faults serviced without any I/O activity. In this case, I/O activity is avoided by reclaiming a page frame from the list of pages awaiting reallocation. **ru_majflt** The number of page faults serviced that required I/O activity. **ru_nswap** The number of times that a process was swapped out of main memory. **ru_inblock** The number of times that the file system performed input. **ru_oublock** The number of times that the file system performed output. **Note:** The numbers that the `ru_inblock` and `ru_oublock` fields display account for real I/O only; data supplied by the caching mechanism is charged only to the first process that reads or writes the data. **ru_msgsnd** The number of IPC messages sent. **ru_msgrcv** The number of IPC messages received. **ru_nsignals** The number of signals delivered. **ru_nvcsw** The number of times a context switch resulted because a process voluntarily gave up the processor before its time slice was completed. This usually occurs while the process waits for a resource to become available. **ru_nivcsw** The number of times a context switch resulted because a higher priority process ran or because the current process exceeded its time slice.

Item	Description
Mode	Indicates which task the subroutine should perform. Acceptable values are as follows: **PTHRDSINFO_RUSAGE_START** Returns the current resource utilization, which will be the start measurement. **PTHRDSINFO_RUSAGE_STOP** Returns total current resource utilization since the last time a **PTHRDSINFO_RUSAGE_START** was performed. If the task **PTHRDSINFO_RUSAGE_START** was not performed, then the resource information returned is the accumulated value since the start of the pthread. **PTHRDSINFO_RUSAGE_COLLECT** Collects resource information for the target thread. If the task **PTHRDSINFO_RUSAGE_START** was not performed, then the resource information returned is the accumulated value since the start of the pthread.

Return Values

Upon successful completion, the **pthread_getrusage_np** subroutine returns a value of 0. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_getrusage_np** subroutine fails if one of the following is true:

Item	Description
EINVAL	The address specified for *RUsage* is NULL, not valid, or a null value for *Ptid* was given.
ESRCH	Either no thread could be found corresponding to the ID thread of the *Ptid* thread or the thread corresponding to the *Ptid* thread ID was not in the current process.

pthread_getschedparam Subroutine

Purpose

Returns the current schedpolicy and schedparam attributes of a thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>
```

```
int pthread_getschedparam ( thread,  schedpolicy,  schedparam)
pthread_t thread;
int *schedpolicy;
struct sched_param *schedparam;
```

Description

The **pthread_getschedparam** subroutine returns the current schedpolicy and schedparam attributes of the thread *thread*. The schedpolicy attribute specifies the scheduling policy of a thread. It may have one of the following values:

Item	Description
SCHED_FIFO	Denotes first-in first-out scheduling.
SCHED_RR	Denotes round-robin scheduling.
SCHED_OTHER	Denotes the default operating system scheduling policy. It is the default value.

The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The `sched_priority` field of the **sched_param** structure contains the priority of the thread. It is an integer value.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in the operating system.

Parameters

Item	Description
thread	Specifies the target thread.
schedpolicy	Points to where the schedpolicy attribute value will be stored.
schedparam	Points to where the schedparam attribute value will be stored.

Return Values

Upon successful completion, the current value of the schedpolicy and schedparam attributes are returned via the *schedpolicy* and *schedparam* parameters, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_getschedparam** subroutine is unsuccessful if the following is true:

Item	Description
ESRCH	The thread *thread* does not exist.

pthread_getspecific or pthread_setspecific Subroutine

Purpose

Returns and sets the thread-specific data associated with the specified key.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
void *pthread_getspecific (key)
pthread_key_t key;
```

```
int pthread_setspecific (key, value)
pthread_key_t key;
const void *value;
```

Description

The **pthread_setspecific** function associates a thread-specific *value* with a *key* obtained via a previous call to **pthread_key_create**. Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

The **pthread_getspecific** function returns the value currently bound to the specified *key* on behalf of the calling thread.

The effect of calling **pthread_setspecific** or **pthread_getspecific** with a *key* value not obtained from **pthread_key_create** or after key has been deleted with **pthread_key_delete** is undefined.

Both **pthread_setspecific** and **pthread_getspecific** may be called from a thread-specific data destructor function. However, calling **pthread_setspecific** from a destructor may result in lost storage or infinite loops.

Parameters

Item	Description
key	Specifies the key to which the value is bound.
value	Specifies the new thread-specific value.

Return Values

The function **pthread_getspecific** returns the thread-specific data value associated with the given key. If no thread-specific data value is associated with key, then the value NULL is returned. If successful, the **pthread_setspecific** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_setspecific** function will fail if:

Item	Description
ENOMEM	Insufficient memory exists to associate the value with the key.

The **pthread_setspecific** function may fail if:

Item	Description
EINVAL	The key value is invalid.

No errors are returned from **pthread_getspecific**.

These functions will not return an error code of EINTR.

pthread_getthrds_np Subroutine

Purpose

Retrieves register and stack information for threads.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_getthrds_np (thread, mode, buf, bufsize, regbuf, regbufsize)
pthread_t *ptid;
int mode;
struct __pthrdsinfo *buf;
int bufsize;
void *regbuf;
int *regbufsize;
```

Description

The **pthread_getthrds_np** subroutine retrieves information on the state of the thread *thread* and its underlying kernel thread, including register and stack information. The thread *thread* must be in suspended state to provide register information for threads.

Parameters

Item	Description
thread	The pointer to the thread. On input it identifies the target thread of the operation, or 0 to operate on the first entry in the list of threads. On output it identifies the next entry in the list of threads, or 0 if the end of the list has been reached. **pthread_getthrds_np** can be used to traverse the whole list of threads by starting with *thread* pointing to 0 and calling **pthread_getthrds_np** repeatedly until it returns with *thread* pointing to 0.

Item	Description

mode	Specifies the type of query. These values can be bitwise or'ed together to specify more than one type of query.
	PTHRDSINFO_QUERY_GPRS
	get general purpose registers
	PTHRDSINFO_QUERY_SPRS
	get special purpose registers
	PTHRDSINFO_QUERY_FPRS
	get floating point registers
	PTHRDSINFO_QUERY_REGS
	get all of the above registers
	PTHRDSINFO_QUERY_TID
	get the kernel thread id
	PTHRDSINFO_QUERY_TLS
	get the thread-local storage information.
	This value can be or'ed with any value of the mode parameter. The thread-local storage information is returned to the caller in a caller-provided buffer, `regbuf`. If the buffer is too small for the data, the buffer is filled up to the end of the buffer and *ERANGE* is returned. The caller also provides the size of the buffer, `regbufsize`, which on return is changed to the size of the thread local storage information even if it does not fit into a buffer.
	The thread-local storage information is returned in form of an array of touplets: memory address and TLS region (unique number assigned by the loader). The TLS region is also included in the loader info structure returned by `loadquery`. If you need any additional information such as TLS size, you can find it in that structure.
	`#typedef struct __pthrdstlsinfo{`
	`  void *pti_vaddr;`
	`  int  pti_region;`
	`} PTHRDS_TLS_INFO;`
	PTHRDSINFO_QUERY_EXTCTX
	get the extended machine context
	PTHRDSINFO_QUERY_ALL
	get everything (except for the extended context, which must be explicitly requested)

Item	Description
buf	Specifies the address of the struct **__pthrdsinfo** structure that will be filled in by **pthread_getthrds_np**. On return, this structure holds the following data (depending on the type of query requested): **__pi_ptid** The thread's thread identifier **__pi_tid** The thread's kernel thread id, or 0 if the thread does not have a kernel thread **__pi_state** The state of the thread, equal to one of the following: **PTHRDSINFO_STATE_RUN** The thread is running **PTHRDSINFO_STATE_READY** The thread is ready to run **PTHRDSINFO_STATE_IDLE** The thread is being initialized **PTHRDSINFO_STATE_SLEEP** The thread is sleeping **PTHRDSINFO_STATE_TERM** The thread is terminated **PTHRDSINFO_STATE_NOTSUP** Error condition **__pi_suspended** 1 if the thread is suspended, 0 if it is not **__pi_returned** The return status of the thread **__pi_ustk** The thread's user stack pointer **__pi_context** The thread's context (register information) If the PTHRDSINFO_QUERY_EXTCTX mode is requested, then the *buf* specifies the address of a `_pthrdsinfox` structure, which, in addition to all of the preceding information, also contains the following: **__pi_ec** The thread's extended context (extended register state)
bufsize	The size of the **__pthrdsinfo** or `__pthrdsinfox` structure in bytes.
regbuf	The location of the buffer to hold the register save data and to pass the TLS information from the kernel if the thread is in a system call.
regbufsize	The pointer to the size of the *regbuf* buffer. On input, it identifies the maximum size of the buffer in bytes. On output, it identifies the number of bytes of register save data and pass the TLS information. If the thread is not in a system call, there is no register save data returned from the kernel, and *regbufsize* is 0. If the size of the register save data is larger than the input value of *regbufsize*, the number of bytes specified by the input value of *regbufsize* is copied to *regbuf*, **pthread_getthrds_np()** returns ERANGE, and the output value of *regbufsize* specifies the number of bytes required to hold all of the register save data.

Return Values

If successful, the **pthread_getthrds_np** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_getthrds_np** function will fail if:

Item	Description
EINVAL	Either *thread* or *buf* is NULL, or *bufsize* is not equal to the size of the **__pthrdsinfo** structure in the library.
ESRCH	No thread could be found corresponding to that specified by the thread ID *thread*.
ERANGE	*regbuf* was not large enough to handle all of the register save data.
ENOMEM	Insufficient memory exists to perform this operation.

pthread_getunique_np Subroutine

Purpose

Returns the sequence number of a thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_getunique_np ( thread,  sequence)
pthread_t *thread;
int *sequence;
```

Description

The **pthread_getunique_np** subroutine returns the sequence number of the thread *thread*. The sequence number is a number, unique to each thread, associated with the thread at creation time.

Note:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_getunique_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

Item	Description
thread	Specifies the thread.
sequence	Points to where the sequence number will be stored.

Return Values

Upon successful completion, the sequence number is returned via the *sequence* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_getunique_np** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *thread* or *sequence* parameters are not valid.
ESRCH	The thread *thread* does not exist.

pthread_join or pthread_detach Subroutine

Purpose

Blocks or detaches the calling thread until the specified thread terminates.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_join (thread, status)
pthread_t thread;
void **status;

int pthread_detach (thread)
pthread_t thread;
```

Description

The **pthread_join** subroutine blocks the calling thread until the thread *thread* terminates. The target thread's termination status is returned in the *status* parameter.

If the target thread is already terminated, but not yet detached, the subroutine returns immediately. It is impossible to join a detached thread, even if it is not yet terminated. The target thread is automatically detached after all joined threads have been woken up.

This subroutine does not itself cause a thread to be terminated. It acts like the **pthread_cond_wait** subroutine to wait for a special condition.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

The **pthread_detach** subroutine is used to indicate to the implementation that storage for the thread whose thread ID is in the location *thread* can be reclaimed when that thread terminates. This storage shall be reclaimed on process exit, regardless of whether the thread has been detached or not, and may include storage for *thread* return value. If *thread* has not yet terminated, **pthread_detach** shall not cause it to terminate. Multiple **pthread_detach** calls on the same target thread causes an error.

Parameters

Item	Description
thread	Specifies the target thread.
status	Points to where the termination status of the target thread will be stored. If the value is **NULL**, the termination status is not returned.

Return Values

If successful, the **pthread_join** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_join** and **pthread_detach** functions will fail if:

Item	Description
EINVAL	The implementation has detected that the value specified by thread does not refer to a joinable thread.
ESRCH	No thread could be found corresponding to that specified by the given thread ID.

The **pthread_join** function will fail if:

Item	Description
EDEADLK	The value of thread specifies the calling thread.

The **pthread_join** function will not return an error code of **EINTR**.

pthread_key_create Subroutine

Purpose

Creates a thread-specific data key.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_key_create ( key, destructor )
pthread_key_t * key;
void (* destructor) (void *);
```

Description

The **pthread_key_create** subroutine creates a thread-specific data key. The key is shared among all threads within the process, but each thread has specific data associated with the key. The thread-specific data is a void pointer, initially set to **NULL**.

The application is responsible for ensuring that this subroutine is called only once for each requested key. This can be done, for example, by calling the subroutine before creating other threads, or by using the one-time initialization facility.

Typically, thread-specific data are pointers to dynamically allocated storage. When freeing the storage, the value should be set to **NULL**. It is not recommended to cast this pointer into scalar data type (**int** for example), because the casts may not be portable, and because the value of **NULL** is implementation dependent.

An optional destructor routine can be specified. It will be called for each thread when it is terminated and detached, after the call to the cleanup routines, if the specific value is not **NULL**. Typically, the destructor routine will release the storage thread-specific data. It will receive the thread-specific data as a parameter.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
key	Points to where the key will be stored.
destructor	Points to an optional destructor routine, used to cleanup data on thread termination. If no cleanup is desired, this pointer should be **NULL**.

Return Values

If successful, the **pthread_key_create** function stores the newly created key value at **key* and returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_key_create** function will fail if:

Item	Description
EAGAIN	The system lacked the necessary resources to create another thread-specific data key, or the system-imposed limit on the total number of keys per process **PTHREAD_KEYS_MAX** has been exceeded.
ENOMEM	Insufficient memory exists to create the key.

The **pthread_key_create** function will not return an error code of **EINTR**.

pthread_key_delete Subroutine

Purpose

Deletes a thread-specific data key.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_key_delete (key)
pthread_key_t key;
```

Description

The **pthread_key_delete** subroutine deletes the thread-specific data key *key*, previously created with the **pthread_key_create** subroutine. The application must ensure that no thread-specific data is associated with the key. No destructor routine is called.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
key	Specifies the key to delete.

Return Values

If successful, the **pthread_key_delete** function returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_key_delete** function will fail if:

Item	Description
EINVAL	The key value is invalid.

The **pthread_key_delete** function will not return an error code of **EINTR**.

pthread_kill Subroutine

Purpose

Sends a signal to the specified thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <signal.h>

int pthread_kill (thread, signal)
pthread_t thread;
int signal;
```

Description

The **pthread_kill** subroutine sends the signal *signal* to the thread *thread*. It acts with threads like the **kill** subroutine with single-threaded processes.

If the receiving thread has blocked delivery of the signal, the signal remains pending on the thread until the thread unblocks delivery of the signal or the action associated with the signal is set to ignore the signal.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
thread	Specifies the target thread for the signal.
signal	Specifies the signal to be delivered. If the signal value is 0, error checking is performed, but no signal is delivered.

Return Values

Upon successful completion, the function returns a value of zero. Otherwise the function returns an error number. If the **pthread_kill** function fails, no signal is sent.

Error Codes

The **pthread_kill** function will fail if:

Item	Description
ESRCH	No thread could be found corresponding to that specified by the given thread ID.
EINVAL	The value of the *signal* parameter is an invalid or unsupported signal number.

The **pthread_kill** function will not return an error code of **EINTR**.

pthread_lock_global_np Subroutine

Purpose

Locks the global mutex.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
void pthread_lock_global_np ()
```

Description

The **pthread_lock_global_np** subroutine locks the global mutex. If the global mutex is currently held by another thread, the calling thread waits until the global mutex is unlocked. The subroutine returns with the global mutex locked by the calling thread.

Use the global mutex when calling a library package that is not designed to run in a multithreaded environment. (Unless the documentation for a library function specifically states that it is compatible with multithreading, assume that it is not compatible; in other words, assume it is nonreentrant.)

The global mutex is one lock. Any code that calls any function that is not known to be reentrant uses the same lock. This prevents dependencies among threads calling library functions and those functions calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. The thread must then call the **pthread_unlock_global_np** subroutine as many times as it called this routine to allow another thread to lock the global mutex.

Note:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_lock_global_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

pthread_mutex_consistent Subroutine

Purpose

Marks the protected state of a robust mutex as consistent.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
int pthread_mutex_consistent(pthread_mutex_t *mutex);
```

Description

The mutex object that is specified by the *mutex* parameter is marked as consistent by calling the **pthread_mutex_consistent** subroutine.

When a thread that holds a robust mutex terminates, the next thread that acquires the mutex is notified about the termination by the **EOWNERDEAD** error code. The mutex is marked as inconsistent and a call to the **pthread_mutex_consistent** subroutine marks the protected state of the robust mutex as consistent.

When a thread that holds a robust mutex terminates when it is in an inconsistent state, the next thread that acquires the mutex is notified about the termination. The robust mutex remains in an inconsistent state. If the **pthread_mutex_consistent** subroutine fails, the state of the robust mutex is not changed.

Parameters

Item	Description
mutex	Specifies the mutex object that must be marked as consistent.

Return Values

On successful completion, the **pthread_mutex_consistent** subroutine returns a value of zero (0). Otherwise, an error code is returned to indicate the error.

Error Codes

The **pthread_mutex_consistent** subroutine can fail because of the following error:

Item	Description
EINVAL	The mutex object that is specified by the *mutex* parameter is not an initialized mutex object, or is not robust, or does not protect an inconsistent state.

The **pthread_mutex_consistent** subroutine does not return the **EINTR** error code.

pthread_mutex_init or pthread_mutex_destroy Subroutine

Purpose

Initializes or destroys a mutex.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutex_init (mutex, attr)
pthread_mutex_t *mutex;
const pthread_mutexattr_t *attr;

int pthread_mutex_destroy (mutex)
pthread_mutex_t *mutex;
```

Description

The **pthread_mutex_init** function initializes the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialization, the state of the mutex becomes initialized and unlocked.

Attempting to initialize an already initialized mutex results in undefined behavior.

The **pthread_mutex_destroy** function destroys the mutex object referenced by mutex; the mutex object becomes, in effect, uninitialized. An implementation may cause **pthread_mutex_destroy** to set the object referenced by *mutex* to an invalid value. A destroyed mutex object can be re-initialized using **pthread_mutex_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

It is safe to destroy an initialized mutex that is unlocked. Attempting to destroy a locked mutex results in undefined behavior.

In cases where default mutex attributes are appropriate, the macro PTHREAD_MUTEX_INITIALIZER can be used to initialize mutexes that are statically allocated. The effect is equivalent to dynamic initialization by a call to **pthread_mutex_init** with parameter *attr* specified as NULL, except that no error checks are performed.

Parameters

Item	Description
mutex	Specifies the mutex to initialize or delete.
attr	Specifies the mutex attributes object.

Return Values

If successful, the **pthread_mutex_init** and **pthread_mutex_destroy** functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks act as if they were performed immediately at the beginning of processing for the function and cause an error return prior to modifying the state of the mutex specified by *mutex*.

Error Codes

The **pthread_mutex_init** function will fail if:

Item	Description
ENOMEM	Insufficient memory exists to initialize the mutex.
EINVAL	The value specified by *attr* is invalid.
EPERM	The caller does not have the privilege to perform the operation in a strictly standards conforming environment where environment variable XPG_SUS_ENV=ON.

The **pthread_mutex_destroy** function may fail if:

Item	Description
EBUSY	The implementation has detected an attempt to destroy the object referenced by *mutex* while it is locked or referenced (for example, while being used in a **pthread_cond_wait**or **pthread_cond_timedwait** by another thread.
EINVAL	The value specified by *mutex* is invalid.

These functions will not return an error code of EINTR.

pthread_mutex_getprioceiling or pthread_mutex_setprioceiling Subroutine

Purpose

Gets and sets the priority ceiling of a mutex.

Syntax

```
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
       int *restrict prioceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
       int prioceiling, int *restrict old_ceiling);
```

Description

The **pthread_mutex_getprioceiling** subroutine returns the current priority ceiling of the mutex.

The **pthread_mutex_setprioceiling** subroutine either locks the mutex if it is unlocked, or blocks until it can successfully lock the mutex, then it changes the mutex's priority ceiling and releases the mutex. When the change is successful, the previous value of the priority ceiling shall be returned in *old_ceiling*. The process of locking the mutex need not adhere to the priority protect protocol.

If the **pthread_mutex_setprioceiling** subroutine fails, the mutex priority ceiling is not changed.

Return Values

If successful, the **pthread_mutex_getprioceiling** and **pthread_mutex_setprioceiling** subroutines return zero; otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_mutex_getprioceiling** and **pthread_mutex_setprioceiling** subroutines fail if the following error codes are returned:

Item	Description
[EINVAL]	The priority requested by the *prioceiling* parameter is out of range.
[EINVAL]	The value specified by the *mutex* parameter does not refer to a currently existing mutex.
[ENOSYS]	This function is not supported (draft 7).
[ENOTSUP]	This function is not supported together with checkpoint/restart.
[EPERM]	The caller does not have the privilege to perform the operation in a strictly standards conforming environment where environment variable XPG_SUS_ENV=ON.

The **pthread_mutex_setprioceiling** subroutine can fail because of one of the following errors:

Item	Description
ENOTRECOVERABLE	The protected state of the mutex cannot be recovered.
EOWNERDEAD	The mutex is a robust mutex, and the process of the thread that owns the mutex terminated while holding the mutex lock.

PTHREAD_MUTEX_INITIALIZER Macro

Purpose

Initializes a static mutex with default attributes.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Description

The **PTHREAD_MUTEX_INITIALIZER** macro initializes the static mutex *mutex*, setting its attributes to default values. This macro should only be used for static mutexes, as no error checking is performed.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

pthread_mutex_lock, pthread_mutex_trylock, or pthread_mutex_unlock Subroutine

Purpose

Locks and unlocks a mutex.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

int pthread_mutex_lock (*mutex*)
pthread_mutex_t ***mutex*;

int pthread_mutex_trylock (*mutex*)
pthread_mutex_t ***mutex*;

int pthread_mutex_unlock (*mutex*)
pthread_mutex_t ***mutex*;

Description

The mutex object referenced by the *mutex* parameter is locked by calling **pthread_mutex_lock**. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by the *mutex* parameter in the locked state with the calling thread as its owner.

If the mutex type is PTHREAD_MUTEX_NORMAL, deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results.

If the mutex type is PTHREAD_MUTEX_ERRORCHECK, then error checking is provided. If a thread attempts to relock a mutex that it has already locked, an error will be returned. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD_MUTEX_RECURSIVE, then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to one. Each time the thread relocks this mutex, the lock count is incremented by one. Each time the thread unlocks the mutex, the lock count is decremented by one. When the lock count reaches zero, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error will be returned.

If the mutex type is PTHREAD_MUTEX_DEFAULT, attempting to recursively lock the mutex results in undefined behavior. Attempting to unlock the mutex if it was not locked by the calling thread results in undefined behavior. Attempting to unlock the mutex if it is not locked results in undefined behavior.

If the mutex is a robust mutex and if the thread that owns the robust mutex terminates while holding the mutex lock, a call to the **pthread_mutex_lock** subroutine returns the **EOWNERDEAD** error code. In this case, the robust mutex is locked by the thread and the protected state of the robust mutex is marked as inconsistent. A call to the **pthread_mutex_consistent** subroutine can be used to mark the protected state of the robust mutex as consistent.

If the mutex is a robust mutex and if the protected state of the robust mutex is inconsistent, a call to the **pthread_mutex_unlock** subroutine marks the state of the robust mutex as permanently unusable. In this case, a call to the **pthread_mutex_destroy** subroutine is the only permissible operation on the robust mutex.

The function **pthread_mutex_trylock** is identical to **pthread_mutex_lock** except that if the robust mutex object referenced by the *mutex* parameter is currently locked (by any thread, including the current thread), the call returns immediately.

The **pthread_mutex_unlock** function releases the mutex object referenced by mutex. The manner in which a mutex is released is dependent upon the mutex's type attribute. If there are threads blocked on the mutex object referenced by the *mutex* parameter when **pthread_mutex_unlock** is called, resulting in the mutex becoming available, the scheduling policy is used to determine which thread will acquire the mutex. (In the case of PTHREAD_MUTEX_RECURSIVE mutexes, the mutex becomes available when the count reaches zero and the calling thread no longer has any locks on this mutex).

If a signal is delivered to a thread waiting for a mutex, upon return from the signal handler the thread resumes waiting for the mutex as if it was not interrupted.

Parameter

Item	Description
mutex	Specifies the mutex to lock.

Return Values

If successful, the **pthread_mutex_lock** and **pthread_mutex_unlock** functions return zero. Otherwise, an error number is returned to indicate the error.

The function **pthread_mutex_trylock** returns zero if a lock on the mutex object referenced by the *mutex* parameter is acquired. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_mutex_trylock** function will fail if:

Item	Description
EBUSY	The mutex could not be acquired because it was already locked.

The **pthread_mutex_lock**, **pthread_mutex_trylock** and **pthread_mutex_unlock** functions will fail if:

Item	Description
EINVAL	The value specified by the *mutex* parameter does not refer to an initialized mutex object.

The **pthread_mutex_lock** function will fail if:

Item	Description
EDEADLK	The current thread already owns the mutex and the mutex type is PTHREAD_MUTEX_ERRORCHECK.

The **pthread_mutex_unlock()** subroutine fails if the following error code is returned:

Item	Description
EPERM	The current thread does not own the mutex, and the type of the mutex is not PTHREAD_MUTEX_NORMAL or the mutex is a robust mutex.

The **pthread_mutex_lock** subroutine or the **pthread_mutex_trylock** subroutine fails if the following error codes are returned:

Item	Description
ENOTRECOVERABLE	The protected state of the mutex cannot be recovered.
EOWNERDEAD	The mutex is a robust mutex, and the process of the thread that owns the mutex terminated while holding the mutex lock.

These subroutines will not return an **EINTR** error code.

pthread_mutex_timedlock Subroutine

Purpose

Locks a mutex (ADVANCED REALTIME).

Syntax

```
#include <pthread.h>
#include <time.h>

int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
       const struct timespec *restrict abs_timeout);
```

Description

The **pthread_mutex_timedlock()** function locks the mutex object referenced by mutex. If the mutex is already locked, the calling thread blocks until the mutex becomes available, as in the **pthread_mutex_lock()** function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait terminates when the specified timeout expires.

The timeout expires when the absolute time specified by *abs_timeout* passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*)—or when the absolute time specified by *abs_timeout* has already been passed at the time of the call.

If the **Timers** option is supported, the timeout is based on the CLOCK_REALTIME clock; if the **Timers** option is not supported, the timeout is based on the system clock as returned by the **time()** function.

The resolution of the timeout matches the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header.

The function never fails with a timeout if the mutex can be locked immediately. The validity of the *abs_timeout* parameter does not need to be checked if the mutex can be locked immediately.

As a consequence of the priority inheritance rules (for mutexes initialized with the PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex adjusts as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

If the mutex is a robust mutex and if the thread that owns the robust mutex terminates while holding the mutex lock, a call to the **pthread_mutex_timedlock** subroutine returns the **EOWNERDEAD** error code. In this case, the robust mutex is locked by the thread and the protected state of the robust mutex is marked as inconsistent. A call to the **pthread_mutex_consistent** subroutine can be used to mark the protected state of the robust mutex as consistent.

If the mutex is a robust mutex and if the protected state of the robust mutex is inconsistent, a call to the **pthread_mutex_unlock** subroutine marks the protected state of the robust mutex as permanently unusable. In this case, a call to the **pthread_mutex_destroy** subroutine is the only permissible operation on the robust mutex.

Application Usage

The **pthread_mutex_timedlock()** function is part of the **Threads** and **Timeouts** options and do not need to be provided on all implementations.

Return Values

If successful, the **pthread_mutex_timedlock** subroutine returns 0; otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_mutex_timedlock** subroutine can fail because of one of the following errors:

Item	Description
EDEADLK	The current thread already owns the mutex.
EINVAL	The mutex was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT, and the calling thread's priority is higher than the mutex's current priority ceiling.
EINVAL	The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.
EINVAL	*abs_timeout* is a NULL pointer.
EINVAL	The value specified by *mutex* does not refer to an initialized mutex object.
ETIMEDOUT	The mutex could not be locked before the specified timeout expired.
ENOTRECOVERABLE	The protected state of the mutex cannot be recovered.
EOWNERDEAD	The mutex is a robust mutex, and the process of the thread that owns the mutex terminated while holding the mutex lock.

This function does not return the **EINTR** error code.

pthread_mutexattr_destroy or pthread_mutexattr_init Subroutine

Purpose

Initializes and destroys mutex attributes.

Library

Threads Library **(libpthreads.a)**

Syntax

```
#include <pthread.h>

int pthread_mutexattr_init (attr)
pthread_mutexattr_t *attr;

int pthread_mutexattr_destroy (attr)
pthread_mutexattr_t *attr;
```

Description

The function **pthread_mutexattr_init** initializes a mutex attributes object *attr* with the default value for all of the attributes defined by the implementation.

The effect of initializing an already initialized mutex attributes object is undefined.

After a mutex attributes object has been used to initialize one or more mutexes, any function affecting the attributes object (including destruction) does not affect any previously initialized mutexes.

The **pthread_mutexattr_destroy** function destroys a mutex attributes object; the object becomes, in effect, uninitialized. An implementation may cause **pthread_mutexattr_destroy** to set the object referenced by *attr* to an invalid value. A destroyed mutex attributes object can be re-initialized using **pthread_mutexattr_init**; the results of otherwise referencing the object after it has been destroyed are undefined.

Parameters

Item	Description
attr	Specifies the mutex attributes object to initialize or delete.

Return Values

Upon successful completion, **pthread_mutexattr_init** and **pthread_mutexattr_destroy** return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_mutexattr_init** function will fail if:

Item	Description
ENOMEM	Insufficient memory exists to initialize the mutex attributes object.

The **pthread_mutexattr_destroy** function will fail if:

Item	Description
EINVAL	The value specified by *attr* is invalid.
	These functions will not return EINTR.

pthread_mutexattr_getkind_np Subroutine

Purpose

Returns the value of the kind attribute of a mutex attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutexattr_getkind_np ( attr,  kind)
pthread_mutexattr_t *attr;
int *kind;
```

Description

The **pthread_mutexattr_getkind_np** subroutine returns the value of the kind attribute of the mutex attributes object *attr*. This attribute specifies the kind of the mutex created with this attributes object. It may have one of the following values:

Item	Description
MUTEX_FAST_NP	Denotes a fast mutex. A fast mutex can be locked only once. If the same thread unlocks twice the same fast mutex, the thread will deadlock. Any thread can unlock a fast mutex. A fast mutex is not compatible with the priority inheritance protocol.
MUTEX_RECURSIVE_NP	Denotes a recursive mutex. A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. The thread must then unlock the mutex as many times as it locked it. Only the thread that locked a recursive mutex can unlock it. A recursive mutex must not be used with condition variables.
MUTEX_NONRECURSIVE_NP	Denotes the default non-recursive POSIX compliant mutex.

Note:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_mutexattr_getkind_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

Item	Description
attr	Specifies the mutex attributes object.
kind	Points to where the kind attribute value will be stored.

Return Values

Upon successful completion, the value of the kind attribute is returned via the *kind* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_mutexattr_getkind_np** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.

pthread_mutexattr_getprioceiling or pthread_mutexattr_setprioceiling Subroutine

Purpose

Gets and sets the prioceiling attribute of the mutex attributes object.

Syntax

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *
       restrict attr, int *restrict prioceiling);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
       int prioceiling);
```

Description

The **pthread_mutexattr_getprioceiling** and **pthread_mutexattr_setprioceiling** subroutines, respectively, get and set the priority ceiling attribute of a mutex attributes object pointed to by the *attr* parameter, which was previously created by the **pthread_mutexattr_init** subroutine.

The *prioceiling* attribute contains the priority ceiling of initialized mutexes. The values of the *prioceiling* parameter are within the maximum range of priorities defined by SCHED_FIFO.

The *prioceiling* parameter defines the priority ceiling of initialized mutexes, which is the minimum priority level at which the critical section guarded by the mutex is executed. In order to avoid priority inversion, the priority ceiling of the mutex is set to a priority higher than or equal to the highest priority of all the threads that may lock that mutex. The values of the *prioceiling* parameter are within the maximum range of priorities defined under the SCHED_FIFO scheduling policy.

Return Values

Upon successful completion, the **pthread_mutexattr_getprioceiling** and **pthread_mutexattr_setprioceiling** subroutines return zero; otherwise, an error number shall be returned to indicate the error.

Error Codes

The **pthread_mutexattr_getprioceiling** and **pthread_mutexattr_setprioceiling** subroutines can fail if:

Item	Description
EINVAL	The value specified by the *attr* or *prioceiling* parameter is invalid.
ENOSYS	This function is not supported (draft 7).
ENOTSUP	This function is not supported together with checkpoint/restart.
EPERM	The caller does not have the privilege to perform the operation in a strictly standards conforming environment where environment variable XPG_SUS_ENV=ON.

pthread_mutexattr_getprotocol or pthread_mutexattr_setprotocol Subroutine

Purpose

Gets and sets the protocol attribute of the mutex attributes object.

Syntax

```
#include <pthread.h>

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *
       restrict attr, int *restrict protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
       int protocol);
```

Description

The **pthread_mutexattr_getprotocol** subroutine and **pthread_mutexattr_setprotocol** subroutine get and set the *protocol* parameter of a mutex attributes object pointed to by the **attr** parameter, which was previously created by the **pthread_mutexattr_init** subroutine.

The protocol attribute defines the protocol to be followed in utilizing mutexes. The value of the *protocol* parameter can be one of the following, which are defined in the **pthread.h** header file:

- **PTHREAD_PRIO_NONE**
- **PTHREAD_PRIO_INHERIT**
- **PTHREAD_PRIO_PROTECT**

When a thread owns a mutex with the **PTHREAD_PRIO_NONE** protocol attribute, its priority and scheduling are not affected by its mutex ownership.

When a thread is blocking higher priority threads because of owning one or more mutexes with the **PTHREAD_PRIO_INHERIT** protocol attribute, it executes at the higher of its priority or the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol.

When a thread owns one or more mutexes initialized with the **PTHREAD_PRIO_PROTECT** protocol, it executes at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute, regardless of whether other threads are blocked on any of these mutexes. Privilege checking is necessary when the mutex priority ceiling is more favored than current thread priority and the thread priority must be changed. The `pthread_mutex_lock` subroutine does not fail because of inappropriate privileges. Locking succeeds in this case, but no boosting is performed.

While a thread is holding a mutex which has been initialized with the **PTHREAD_PRIO_INHERIT** or **PTHREAD_PRIO_PROTECT** protocol attributes, it is not subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed, such as by a call to the **sched_setparam** subroutine. Likewise, when a thread unlocks a mutex that has been initialized with the **PTHREAD_PRIO_INHERIT** or **PTHREAD_PRIO_PROTECT** protocol attributes, it is not subject to being moved to the tail of the scheduling queue at its priority in the event that its original priority is changed.

If a thread simultaneously owns several mutexes initialized with different protocols, it executes at the highest of the priorities that it would have obtained by each of these protocols.

When a thread makes a call to the **pthread_mutex_lock** subroutine, the mutex was initialized with the protocol attribute having the value **PTHREAD_PRIO_INHERIT**, when the calling thread is blocked because the mutex is owned by another thread, that owner thread inherits the priority level of the calling thread as long as it continues to own the mutex. The implementation updates its execution priority to the maximum of its assigned priority and all its inherited priorities. Furthermore, if this owner thread itself becomes blocked on another mutex, the same priority inheritance effect shall be propagated to this other owner thread, in a recursive manner.

Return Values

Upon successful completion, the **pthread_mutexattr_getprotocol** subroutine and the **pthread_mutexattr_setprotocol** subroutine return zero; otherwise, an error number shall be returned to indicate the error.

Error Codes

The **pthread_mutexattr_setprotocol** subroutine fails if:

Item	Description
ENOTSUP	The value specified by the *protocol* parameter is an unsupported value.

The **pthread_mutexattr_getprotocol** subroutine and **pthread_mutexattr_setprotocol** subroutine can fail if:

Item	Description
EINVAL	The value specified by the *attr* parameter or the *protocol* parameter is invalid.
ENOSYS	This function is not supported (draft 7).
ENOTSUP	This function is not supported together with checkpoint/restart.
EPERM	The caller does not have the privilege to perform the operation in a strictly standards conforming environment where environment variable XPG_SUS_ENV=ON.

pthread_mutexattr_getrobust and pthread_mutexattr_setrobust Subroutine

Purpose

Gets and sets the *robust* attribute of the mutex attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
int pthread_mutexattr_getrobust(const pthread_mutexattr_t *restrict attr, int *restrict robust);
int pthread_mutexattr_setrobust(pthread_mutexattr_t *attr, int  robust);
```

Description

The **pthread_mutexattr_getrobust** subroutine obtains the value of the *robust* attribute from the attributes object that is specified by the *attr* parameter. The **pthread_mutexattr_setrobust** subroutine sets the value of the *robust* attribute in an initialized attributes object that is specified by the *attr* parameter.

The *robust* attribute can have the value **PTHREAD_MUTEX_STALLED** or **PTHREAD_MUTEX_ROBUST**, and these values are defined in the **pthread.h** header file. The default value is **PTHREAD_MUTEX_STALLED**.

When a thread that holds the mutex terminates while the *robust* attribute is set to **PTHREAD_MUTEX_STALLED**, and another thread attempts to acquire the mutex, no action is performed.

When a thread that holds the mutex terminates while the *robust* attribute is set to **PTHREAD_MUTEX_ROBUST**, and the process-shared attribute is set to **PTHREAD_PROCESS_SHARED**, the next thread that attempts to get the mutex is notified about the termination. The notified thread becomes the new mutex owner and the protected state of the mutex is now marked as inconsistent.

When the protected state of a robust mutex is inconsistent, the **pthread_mutex_consistent** subroutine can be used to mark the protected state of the robust mutex as consistent.

When the protected state of a robust mutex is inconsistent, a call to the **pthread_mutex_unlock** subroutine, without a call to the **pthread_mutex_consistent** subroutine, marks the protected state of the robust mutex as permanently unusable. In this case, a call to the **pthread_mutex_destroy** subroutine is the only permissible operation on the robust mutex.

Parameters

Item	Description
attr	Specifies the mutex attributes object.

Item	Description
robust	Indicates the object that stores the value of the *robust* attribute.

Return Values

On successful completion, the **pthread_mutexattr_setrobust** subroutine returns a value of zero (0). Otherwise, an error code is returned to indicate the error.

On successful completion, the **pthread_mutexattr_getrobust** subroutine returns a value of zero (0). The subroutine stores the value of the *robust* attribute for the *attr* parameter into an object that is specified by the *robust* attribute. Otherwise, an error code is returned to indicate the error.

Error Codes

The **pthread_mutexattr_getrobust** subroutine or the **pthread_mutexattr_setrobust** subroutine can fail because of the following error:

Item	Description
EINVAL	The value that is specified by the *attr* parameter is invalid. For the **pthread_mutexattr_setrobust** subroutine, this error code can also mean that the new value that is specified for the robust attribute is outside the range of permissible values.

The **pthread_mutexattr_getrobust** subroutine or the **pthread_mutexattr_setrobust** subroutine does not return the EINTR error code.

pthread_mutexattr_getpshared or pthread_mutexattr_setpshared Subroutine

Purpose

Sets and gets process-shared attribute.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutexattr_getpshared (attr, pshared)
const pthread_mutexattr_t *attr;
int *pshared;

int pthread_mutexattr_setpshared (attr, pshared)
pthread_mutexattr_t *attr;
int pshared;
```

Description

The **pthread_mutexattr_getpshared** subroutine obtains the value of the process-shared attribute from the attributes object referenced by *attr*. The **pthread_mutexattr_setpshared** subroutine is used to set the process-shared attribute in an initialized attributes object referenced by *attr*.

The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the **process-shared** attribute is PTHREAD_PROCESS_PRIVATE, the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

Parameters

Item	Description
attr	Specifies the mutex attributes object.
pshared	Points to where the pshared attribute value will be stored.

Return Values

Upon successful completion, the **pthread_mutexattr_setpshared** subroutine returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the **pthread_mutexattr_getpshared** subroutine returns zero and stores the value of the process-shared attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_mutexattr_getpshared** and **pthread_mutexattr_setpshared** subroutines will fail if:

Item	Description
EINVAL	The value specified by *attr* is invalid.

The **pthread_mutexattr_setpshared** function will fail if:

Item	Description
EINVAL	The new value specified for the attribute is outside the range of legal values for that attribute.

These subroutines will not return an error code of EINTR.

pthread_mutexattr_gettype or pthread_mutexattr_settype Subroutine

Purpose

Gets or sets a mutex type.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutexattr_gettype (attr, type)
const pthread_mutexattr_t *attr;
int *type;

int pthread_mutexattr_settype (attr, type)
```

```
pthread_mutexattr_t *attr;
int type;
```

Description

The **pthread_mutexattr_gettype** and **pthread_mutexattr_settype** subroutines respectively get and set the mutex type attribute. This attribute is set in the *type* parameter to these subroutines. The default value of the type attribute is PTHREAD_MUTEX_DEFAULT. The type of mutex is contained in the type attribute of the mutex attributes. Valid mutex types include:

Item	Description
PTHREAD_MUTEX_NORMAL	This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking it will deadlock. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.
PTHREAD_MUTEX_ERRORCHECK	This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking it will return with an error. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
PTHREAD_MUTEX_RECURSIVE	A thread attempting to relock this mutex without first unlocking it will succeed in locking the mutex. The relocking deadlock which can occur with mutexes of type PTHREAD_MUTEX_NORMAL cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex which another thread has locked will return with an error. A thread attempting to unlock an unlocked mutex will return with an error.
PTHREAD_MUTEX_DEFAULT	Attempting to recursively lock a mutex of this type results in undefined behavior. Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behavior. Attempting to unlock a mutex of this type which is not locked results in undefined behavior. An implementation is allowed to map this mutex to one of the other mutex types.

It is advised that an application should not use a PTHREAD_MUTEX_RECURSIVE mutex with condition variables because the implicit unlock performed for a **pthread_cond_wait** or **pthread_cond_timedwait** may not actually release the mutex (if it had been locked multiple times). If this happens, no other thread can satisfy the condition of the predicate.

Parameters

Item	Description
attr	Specifies the mutex object to get or set.
type	Specifies the type to get or set.

Return Values

If successful, the **pthread_mutexattr_settype** subroutine returns zero. Otherwise, an error number is returned to indicate the error. Upon successful completion, the **pthread_mutexattr_gettype** subroutine returns zero and stores the value of the type attribute of *attr* into the object referenced by the *type* parameter. Otherwise an error is returned to indicate the error.

Error Codes

The **pthread_mutexattr_gettype** and **pthread_mutexattr_settype** subroutines will fail if:

Item	Description
EINVAL	The value of the *type* parameter is invalid.
EINVAL	The value specified by the *attr* parameter is invalid.

pthread_mutexattr_setkind_np Subroutine

Purpose

Sets the value of the kind attribute of a mutex attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_mutexattr_setkind_np ( attr,  kind)
pthread_mutexattr_t *attr;
int kind;
```

Description

The **pthread_mutexattr_setkind_np** subroutine sets the value of the kind attribute of the mutex attributes object *attr*. This attribute specifies the kind of the mutex created with this attributes object.

Note:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_mutexattr_setkind_np** subroutine is not portable.

This subroutine is provided only for compatibility with the DCE threads. It should not be used when writing new applications.

Parameters

Item	Description
attr	Specifies the mutex attributes object.
kind	Specifies the kind to set. It must have one of the following values: **MUTEX_FAST_NP** Denotes a fast mutex. A fast mutex can be locked only once. If the same thread unlocks twice the same fast mutex, the thread will deadlock. Any thread can unlock a fast mutex. A fast mutex is not compatible with the priority inheritance protocol. **MUTEX_RECURSIVE_NP** Denotes a recursive mutex. A recursive mutex can be locked more than once by the same thread without causing that thread to deadlock. The thread must then unlock the mutex as many times as it locked it. Only the thread that locked a recursive mutex can unlock it. A recursive mutex must not be used with condition variables. **MUTEX_NONRECURSIVE_NP** Denotes the default non-recursive POSIX compliant mutex.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_mutexattr_setkind_np** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *attr* parameter is not valid.
ENOTSUP	The value of the *kind* parameter is not supported.

pthread_once Subroutine

Purpose

Executes a routine exactly once in a process.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_once (once_control, init_routine)
pthread_once_t *once_control;
void (*init_routine)(void);
```

```

```

```
,
```

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

Description

The **pthread_once** subroutine executes the routine *init_routine* exactly once in a process. The first call to this subroutine by any thread in the process executes the given routine, without parameters. Any subsequent call will have no effect.

The *init_routine* routine is typically an initialization routine. Multiple initializations can be handled by multiple instances of **pthread_once_t** structures. This subroutine is useful when a unique initialization has to be done by one thread among many. It reduces synchronization requirements.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Parameters

Item	Description
once_control	Points to a synchronization control structure. This structure has to be initialized by the static initializer macro **PTHREAD_ONCE_INIT**.
init_routine	Points to the routine to be executed.

Return Values

Upon successful completion, **pthread_once** returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

No errors are defined. The **pthread_once** function will not return an error code of EINTR.

PTHREAD_ONCE_INIT Macro

Purpose

Initializes a once synchronization control structure.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
static pthread_once_t once_block = PTHREAD_ONCE_INIT;
```

Description

The **PTHREAD_ONCE_INIT** macro initializes the static once synchronization control structure *once_block*, used for one-time initializations with the **pthread_once** subroutine. The once synchronization control structure must be static to ensure the unicity of the initialization.

Note: The **pthread.h** file header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

pthread_rwlock_init or pthread_rwlock_destroy Subroutine

Purpose

Initializes or destroys a read-write lock object.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlock_init (rwlock, attr)
pthread_rwlock_t *rwlock;
const pthread_rwlockattr_t *attr;

int pthread_rwlock_destroy (rwlock)
pthread_rwlock_t *rwlock;
pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

Description

The **pthread_rwlock_init** subroutine initializes the read-write lock referenced by *rwlock* with the attributes referenced by *attr*. If *attr* is NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Once initialized, the lock can be used any number of times without being re-initialized. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked. Results are undefined if **pthread_rwlock_init** is called specifying an already initialized read-write lock. Results are undefined if a read-write lock is used without first being initialized.

If the **pthread_rwlock_init** function fails, *rwlock* is not initialized and the contents of *rwlock* are undefined.

The **pthread_rwlock_destroy** function destroys the read-write lock object referenced by *rwlock* and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is re-initialized by another call to **pthread_rwlock_init**. An implementation may cause **pthread_rwlock_destroy** to set the object referenced by *rwlock* to an invalid value. Results are undefined if **pthread_rwlock_destroy** is called when any thread holds *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behavior. A destroyed read-write lock object can be re-initialized using **pthread_rwlock_init**; the results of otherwise referencing the read-write lock object after it has been destroyed are undefined.

In cases where default read-write lock attributes are appropriate, the macro **PTHREAD_RWLOCK_INITIALIZER** can be used to initialize read-write locks that are statically allocated. The effect is equivalent to dynamic initialization by a call to **pthread_rwlock_init** with the parameter *attr* specified as NULL, except that no error checks are performed.

Parameters

Item	Description
rwlock	Specifies the read-write lock to be initialized or destroyed.
attr	Specifies the attributes of the read-write lock to be initialized.

Return Values

If successful, the **pthread_rwlock_init** and **pthread_rwlock_destroy** functions return zero. Otherwise, an error number is returned to indicate the error. The EBUSY and EINVAL error checks, if implemented,

will act as if they were performed immediately at the beginning of processing for the function and caused an error return prior to modifying the state of the read-write lock specified by *rwlock*.

Error Codes

The **pthread_rwlock_init** subroutine will fail if:

Item	Description
ENOMEM	Insufficient memory exists to initialize the read-write lock.
EINVAL	The value specified by *attr* is invalid.

The **pthread_rwlock_destroy** subroutine will fail if:

Item	Description
EBUSY	The implementation has detected an attempt to destroy the object referenced by *rwlock* while it is locked.
EINVAL	The value specified by *attr* is invalid.

pthread_rwlock_rdlock or pthread_rwlock_tryrdlock Subroutines

Purpose

Locks a read-write lock object for reading.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlock_rdlock (rwlock)
pthread_rwlock_t *rwlock;

int pthread_rwlock_tryrdlock (rwlock)
pthread_rwlock_t *rwlock;
```

Description

The **pthread_rwlock_rdlock** function applies a read lock to the read-write lock referenced by *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the **pthread_rwlock_rdlock call**) until it can acquire the lock. Results are undefined if the calling thread holds a write lock on *rwlock* at the time the call is made.

Implementations are allowed to favor writers over readers to avoid writer starvation.

A thread may hold multiple concurrent read locks on *rwlock* (that is, successfully call the **pthread_rwlock_rdlock** function *n* times). If so, the thread must perform matching unlocks (that is, it must call the **pthread_rwlock_unlock** function *n* times).

The function **pthread_rwlock_tryrdlock** applies a read lock as in the **pthread_rwlock_rdlock** function with the exception that the function fails if any thread holds a write lock on *rwlock* or there are writers blocked on *rwlock*.

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

Parameters

Item	Description
rwlock	Specifies the read-write lock to be locked for reading.

Return Values

If successful, the **pthread_rwlock_rdlock** function returns zero. Otherwise, an error number is returned to indicate the error.

The function **pthread_rwlock_tryrdlock** returns zero if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the error.

Error Codes

The **pthread_rwlock_tryrdlock** function will fail if:

Item	Description
EBUSY	The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

The **pthread_rwlock_rdlock** and **pthread_rwlock_tryrdlock** functions will fail if:

Item	Description
EINVAL	The value specified by *rwlock* does not refer to an initialized read-write lock object.
EDEADLK	The current thread already owns the read-write lock for writing.
EAGAIN	The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded.

Implementation Specifics

Realtime applications may encounter priority inversion when using read-write locks. The problem occurs when a high priority thread 'locks' a read-write lock that is about to be 'unlocked' by a low priority thread, but the low priority thread is preempted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period of time. During system design, realtime programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by read-write locks execute at a high priority, so that a thread cannot be preempted while executing in its critical section.

pthread_rwlock_attr_setfavorwriters_np or pthread_rwlock_attr_getfavorwriters_np Subroutine

Purpose

Sets or returns a read/write lock attribute that enables the pthread library to specify the preference while scheduling the threads to get the read/write lock in write mode.

Library

Threads library (`libthreads.a`)

Syntax

```
#define PTHREAD_RWLOCK_FAVORREADERS 0
#define PTHREAD_RWLOCK_FAVORWRITERS 1
#include <pthread.h>

int pthread_rwlock_attr_setfavorwriters_np(pthread_rwlockattr_t *user_attribute_structure,
int favor_attribute)

int pthread_rwlock_attr_getfavorwrites_np(pthread_rwlockattr_t *user_attribute_structure,
int *return_attribute)
```

Description

The **pthread_rwlock_attr_setfavorwriters_np** subroutine can be used by an application to initialize the attributes of a read/write lock. You can specify the pthread library to prioritize the scheduling of the threads that requires the read/write lock in write mode. When the pthread library schedules the writer-threads (threads that write data) to get the read/write lock in write mode, the pthread library does not support recursion by threads that are holding a the read/write lock in read mode. Unexpected results can occur when the threads hold a read/write lock in the read mode more than once.

The **pthread_rwlock_attr_setfavorwriters_np** subroutine sets an attribute to choose writer-threads or reader-threads (threads that need to read data) depending on the value of the *favor_attribute* attribute that is specified in the subroutine. When the *favor_attribute* attribute is passed to the **pthread_rwlock_init** subroutine, the initialized read/write lock considers the specified preference in the attribute structure.

The **pthread_rwlock_attr_getfavorwriters_np** subroutine returns the current preference that is set in the read/write lock attribute structure. By default, the reader-threads are preferred over writer-threads to get a read/write lock.

Parameters

favor_attribute
Indicates the pthread library to prioritize a writer-thread or a reader-thread to get a read/write lock.

return_attribute
Returns the specified value in the *favor_attribute* parameter from the attribute structure of the read/write lock.

Return Values

If the operation is successful, the **pthread_rwlock_attr_setfavorwriters_np** and **pthread_rwlock_attr_getfavorwriters_np** subroutines return zero. Otherwise, a number is returned to indicate the error.

Error Codes

ENOMEM
Insufficient memory to initialize the read/write lock attributes object.

EINVAL
Invalid parameters.

pthread_rwlock_timedrdlock Subroutine

Purpose

Locks a read-write lock for reading.

Syntax

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
       const struct timespec *restrict abs_timeout);
```

Description

The **pthread_rwlock_timedrdlock()** function applies a read lock to the read-write lock referenced by *rwlock* as in the **pthread_rwlock_rdlock()** function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait terminates when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*)—or when the absolute time specified by *abs_timeout* has already been passed at the time of the call.

If the **Timers** option is supported, the timeout is based on the CLOCK_REALTIME clock; if the **Timers** option is not supported, the timeout is based on the system clock as returned by the **time()** function.

The resolution of the timeout matches the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header.

The function never fails with a timeout if the lock can be acquired immediately. The validity of the *abs_timeout* parameter does not need to be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread that is blocked on a read-write lock through a call to **pthread_rwlock_timedrdlock()**, the thread resumes waiting for the lock (as if it were not interrupted) after the signal handler returns.

The calling thread can deadlock if it holds a write lock on **rwlock** at the time the call is made. The results are undefined if this function is called with an uninitialized read-write lock.

Application Usage

The **pthread_rwlock_timedrdlock()** function is part of the **Threads** and **Timeouts** options and do not need to be provided on all implementations.

Return Values

The **pthread_rwlock_timedrdlock()** function returns 0 if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_rwlock_timedrdlock()** function fails if:

Item	Description
[ETIMEDOUT]	The lock could not be acquired before the specified timeout expired.

The **pthread_rwlock_timedrdlock()** function might fail if:

Item	Description
[EAGAIN]	The read lock could not be acquired because the maximum number of read locks for lock would be exceeded.
[EDEADLK]	The calling thread already holds a write lock on *rwlock*.
[EINVAL]	The value specified by *rwlock* does not refer to an initialized read-write lock object, or the *abs_timeout* nanosecond value is less than 0 or greater than or equal to 1000 million.

This function does not return an error code of [EINTR].

pthread_rwlock_timedwrlock Subroutine

Purpose

Locks a read-write lock for writing.

Syntax

```
#include <pthread.h>
#include <time.h>

int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
       const struct timespec *restrict abs_timeout);
```

Description

The **pthread_rwlock_timedwrlock()** function applies a write lock to the read-write lock referenced by *rwlock* as in the **pthread_rwlock_wrlock()** function. However, if the lock cannot be acquired without waiting for other threads to unlock the lock, this wait terminates when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds *abs_timeout*)—or when the absolute time specified by *abs_timeout* has already been passed at the time of the call.

If the **Timers** option is supported, the timeout is based on the CLOCK_REALTIME clock; if the **Timers** option is not supported, the timeout is based on the system clock as returned by the **time()** function.

The resolution of the timeout matches the resolution of the clock on which it is based. The **timespec** data type is defined in the **<time.h>** header.

The function never fails with a timeout if the lock can be acquired immediately. The validity of the *abs_timeout* parameter does not need to be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread that is blocked on a read-write lock through a call to **pthread_rwlock_timedwrlock()**, the thread resumes waiting for the lock (as if it were not interrupted) after the signal handler returns.

The calling thread can deadlock if it holds the read-write lock at the time the call is made. The results are undefined if this function is called with an uninitialized read-write lock.

Application Usage

The **pthread_rwlock_timedwrlock()** function is part of the **Threads** and **Timeouts** options and do not need to be provided on all implementations.

Return Values

The **pthread_rwlock_timedwrlock()** function returns 0 if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_rwlock_timedrdlock()** function fails if:

Item	Description
ETIMEDOUT	The lock could not be acquired before the specified timeout expired.

The **pthread_rwlock_timedrdlock()** function might fail if:

Item	Description
EDEADLK	The calling thread already holds the *rwlock*.
EINVAL	The value specified by *rwlock* does not refer to an initialized read-write lock object, or the *abs_timeout* nanosecond value is less than 0 or greater than or equal to 1000 million.

This function does not return an error code of EINTR.

pthread_rwlock_unlock Subroutine

Purpose

Unlocks a read-write lock object.

Library

Threads Library (**libthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlock_unlock (rwlock)
pthread_rwlock_t *rwlock;
```

Description

The **pthread_rwlock_unlock** subroutine is called to release a lock held on the read-write lock object referenced by *rwlock*. Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

If this subroutine is called to release a read lock from the read-write lock object and there are other read locks currently held on this read-write lock object, the read-write lock object remains in the read locked state. If this subroutine releases the calling thread's last read lock on this read-write lock object, then the calling thread is no longer one of the owners of the object. If this subroutine releases the last read lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If this subroutine is called to release a write lock for this read-write lock object, the read-write lock object will be put in the unlocked state with no owners.

If the call to the **pthread_rwlock_unlock** subroutine results in the read-write lock object becoming unlocked and there are multiple threads waiting to acquire the read-write lock object for writing, the scheduling policy is used to determine which thread acquires the read-write lock object for writing. If there are multiple threads waiting to acquire the read-write lock object for reading, the scheduling policy is used to determine the order in which the waiting threads acquire the read-write lock object for reading. If there are multiple threads blocked on *rwlock* for both read locks and write locks, it is unspecified whether the readers acquire the lock first or whether a writer acquires the lock first.

Results are undefined if any of these subroutines are called with an uninitialized read-write lock.

Parameters

Item	Description
rwlock	Specifies the read-write lock to be unlocked.

Return Values

If successful, the **pthread_rwlock_unlock** subroutine returns zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_rwlock_unlock** subroutine may fail if:

Item	Description
EINVAL	The value specified by *rwlock* does not refer to an initialized read-write lock object.
EPERM	The current thread does not own the read-write lock.

pthread_rwlock_wrlock or pthread_rwlock_trywrlock Subroutines

Purpose

Locks a read-write lock object for writing.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlock_wrlock (rwlock)
pthread_rwlock_t *rwlock;

int pthread_rwlock_trywrlock (rwlock)
pthread_rwlock_t *rwlock;
```

Description

The **pthread_rwlock_wrlock** subroutine applies a write lock to the read-write lock referenced by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rwlock*. Otherwise, the thread blocks (that is, does not return from the **pthread_rwlock_wrlock** call) until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

Implementations are allowed to favor writers over readers to avoid writer starvation.

The **pthread_rwlock_trywrlock** subroutine applies a write lock like the **pthread_rwlock_wrlock** subroutine, with the exception that the function fails if any thread currently holds *rwlock* (for reading or writing).

Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

Real-time applications may encounter priority inversion when using read-write locks. The problem occurs when a high priority thread 'locks' a read-write lock that is about to be 'unlocked' by a low priority thread,

but the low priority thread is pre-empted by a medium priority thread. This scenario leads to priority inversion; a high priority thread is blocked by lower priority threads for an unlimited period. During system design, real-time programmers must take into account the possibility of this kind of priority inversion. They can deal with it in a number of ways, such as by having critical sections that are guarded by read-write locks execute at a high priority, so that a thread cannot be pre-empted while executing in its critical section.

Note: With a large number of readers and relatively few writers there is a possibility of writer starvation. If the threads are waiting for an exclusive write lock on the read-write lock, and there are threads that currently hold a shared read lock, the subsequent attempts to acquire a shared read lock request are granted, where as the attempts to acquire an exclusive write lock waits.

Parameters

Item	Description
rwlock	Specifies the read-write lock to be locked for writing.

Return Values

If successful, the **pthread_rwlock_wrlock** subroutine returns zero. Otherwise, an error number is returned to indicate the error.

The **pthread_rwlock_trywrlock** subroutine returns zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise an error number is returned to indicate the error.

Error Codes

The **pthread_rwlock_trywrlock** subroutine will fail if:

Item	Description
EBUSY	The read-write lock could not be acquired for writing because it was already locked for reading or writing.

The **pthread_rwlock_wrlock** and **pthread_rwlock_trywrlock** subroutines may fail if:

Item	Description
EINVAL	The value specified by *rwlock* does not refer to an initialized read-write lock object.
EDEADLK	The current thread already owns the read-write lock for writing or reading.

pthread_rwlockattr_init or pthread_rwlockattr_destroy Subroutines

Purpose

Initializes and destroys read-write lock attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlockattr_init (attr)
pthread_rwlockattr_t *attr;
```

```
int pthread_rwlockattr_destroy (attr)
pthread_rwlockattr_t *attr;
```

Description

The **pthread_rwlockattr_init** subroutine initializes a read-write lock attributes object *attr* with the default value for all of the attributes defined by the implementation. Results are undefined if **pthread_rwlockattr_init** is called specifying an already initialized read-write lock attributes object.

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) does not affect any previously initialized read-write locks.

The **pthread_rwlockattr_destroy** subroutine destroys a read-write lock attributes object. The effect of subsequent use of the object is undefined until the object is re-initialized by another call to **pthread_rwlockattr_init**. An implementation may cause **pthread_rwlockattr_destroy** to set the object referenced by *attr* to an invalid value.

Parameters

Item	Description
attr	Specifies a read-write lock attributes object to be initialized or destroyed.

Return Value

If successful, the **pthread_rwlockattr_init** and **pthread_rwlockattr_destroy** subroutines return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_rwlockattr_init** subroutine will fail if:

Item	Description
ENOMEM	Insufficient memory exists to initialize the read-write lock attributes object.

The **pthread_rwlockattr_destroy** subroutine will fail if:

Item	Description
EINVAL	The value specified by *attr* is invalid.

pthread_rwlockattr_getpshared or pthread_rwlockattr_setpshared Subroutines

Purpose

Gets and sets process-shared attribute of read-write lock attributes object.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_rwlockattr_getpshared (attr, pshared)
```

```
const pthread_rwlockattr_t *attr;
int *pshared;

int pthread_rwlockattr_setpshared (attr, pshared)
pthread_rwlockattr_t *attr;
int pshared;
```

Description

The process-shared attribute is set to PTHREAD_PROCESS_SHARED to permit a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the read-write lock will only be operated upon by threads created within the same process as the thread that initialized the read-write lock; if threads of differing processes attempt to operate on such a read-write lock, the behavior is undefined. The default value of the process-shared attribute is PTHREAD_PROCESS_PRIVATE.

The **pthread_rwlockattr_getpshared** subroutine obtains the value of the process-shared attribute from the initialized attributes object referenced by *attr*. The **pthread_rwlockattr_setpshared** subroutine is used to set the process-shared attribute in an initialized attributes object referenced by *attr*.

Parameters

Item	Description
attr	Specifies the initialized attributes object.
pshared	Specifies the process-shared attribute of read-write lock attributes object to be gotten and set.

Return Values

If successful, the **pthread_rwlockattr_setpshared** subroutine returns zero. Otherwise, an error number is returned to indicate the error.

Upon successful completion, the **pthread_rwlockattr_getpshared** subroutine returns zero and stores the value of the process-shared attribute of *attr* into the object referenced by the *pshared* parameter. Otherwise an error number is returned to indicate the error.

Error Codes

The **pthread_rwlockattr_getpshared** and **pthread_rwlockattr_setpshared** subroutines will fail if:

Item	Description
EINVAL	The value specified by *attr* is invalid.

The **pthread_rwlockattr_setpshared** subroutine will fail if:

Item	Description
EINVAL	The new value specified for the attribute is outside the range of legal values for that attribute.

pthread_self Subroutine

Purpose

Returns the calling thread's ID.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
pthread_t pthread_self (void);
```

Description

The **pthread_self** subroutine returns the calling thread's ID.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

Return Values

The calling thread's ID is returned.

Errors

No errors are defined.

The **pthread_self** function will not return an error code of EINTR.

pthread_setcancelstate, pthread_setcanceltype, or pthread_testcancel Subroutines

Purpose

Sets the calling thread's cancelability state.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

int pthread_setcancelstate (state, oldstate)
int state;
int *oldstate;

int pthread_setcanceltype (type, oldtype)
int type;
int *oldtype;

int pthread_testcancel (void)
```

Description

The **pthread_setcancelstate** subroutine atomically both sets the calling thread's cancelability state to the indicated state and returns the previous cancelability state at the location referenced by *oldstate*. Legal values for state are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE.

The **pthread_setcanceltype** subroutine atomically both sets the calling thread's cancelability type to the indicated type and returns the previous cancelability type at the location referenced by *oldtype*. Legal values for type are PTHREAD_CANCEL_DEFERRED and PTHREAD_CANCEL_ASYNCHRONOUS.

The cancelability state and type of any newly created threads, including the thread in which **main** was first invoked, are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DEFERRED respectively.

The **pthread_testcancel** subroutine creates a cancellation point in the calling thread. The **pthread_testcancel** subroutine has no effect if cancelability is disabled.

Parameters

Item	Description
state	Specifies the new cancelability state to set. It must have one of the following values: **PTHREAD_CANCEL_DISABLE** Disables cancelability; the thread is not cancelable. Cancellation requests are held pending. **PTHREAD_CANCEL_ENABLE** Enables cancelability; the thread is cancelable, according to its cancelability type. This is the default value.
oldstate	Points to where the previous cancelability state value will be stored.
type	Specifies the new cancelability type to set.
oldtype	Points to where the previous cancelability type value will be stored.

Return Values

If successful, the **pthread_setcancelstate** and **pthread_setcanceltype** subroutines return zero. Otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_setcancelstate** subroutine will fail if:

Item	Description
EINVAL	The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

The **pthread_setcanceltype** subroutine will fail if:

Item	Description
EINVAL	The specified type is not PTHREAD_CANCEL_DEFERRED or PTHREAD_CANCEL_ASYNCHRONOUS.

These subroutines will not return an error code of EINTR.

pthread_setschedparam Subroutine

Purpose

Sets `schedpolicy` and `schedparam` attributes of a thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <sys/sched.h>

int pthread_setschedparam (thread, schedpolicy, schedparam)
pthread_t thread;
int schedpolicy;
const struct sched_param *schedparam;
```

Description

The **pthread_setschedparam** subroutine dynamically sets the schedpolicy and schedparam attributes of the thread *thread*. The schedpolicy attribute specifies the scheduling policy of the thread. The schedparam attribute specifies the scheduling parameters of a thread created with this attributes object. The `sched_priority` field of the **sched_param** structure contains the priority of the thread. It is an integer value.

If the target thread has system contention scope, the process must have root authority to set the scheduling policy to either **SCHED_FIFO** or **SCHED_RR**.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

This subroutine is part of the Base Operating System (BOS) Runtime. The implementation of this subroutine is dependent on the priority scheduling POSIX option. The priority scheduling POSIX option is implemented in the operating system.

Parameters

Item	Description
thread	Specifies the target thread.
schedpolicy	Points to the schedpolicy attribute to set. It must have one of the following values: **SCHED_FIFO** Denotes first-in first-out scheduling. **SCHED_RR** Denotes round-robin scheduling. **SCHED_OTHER** Denotes the default operating system scheduling policy. It is the default value. If *schedpolicy* is SCHED_OTHER, then *sched_priority* must be in the range from 40 to 80, where 40 is the least favored priority and 80 is the most favored. **Note:** Priority of threads with a process contention scope and a SCHED_OTHER policy is controlled by the kernel; thus, setting the priority of such a thread has no effect. However, priority of threads with a system contention scope and a SCHED_OTHER policy can be modified. The modification directly affects the underlying kernel thread **nice** value.

Item	Description
schedparam	Points to where the scheduling parameters to set are stored. The `sched_priority` field must be in the range from 1 to 127, where 1 is the least favored priority, and 127 the most favored. If *schedpolicy* is SCHED_OTHER, then *sched_priority* must be in the range from 40 to 80, where 40 is the least favored priority and 80 is the most favored. Users can change the priority of a thread when setting its scheduling policy to SCHED_OTHER. The legal values that can be passed to `pthread_setschedparam` range from 40 to 80. Only privileged users can set a priority higher than 60. A value ranging from 1 to 39 provides the same priority as 40, and a value ranging from 81 to 127 provides the same priority as 80.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_setschedparam** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *thread* or *schedparam* parameters are not valid.
ENOSYS	The priority scheduling POSIX option is not implemented.
ENOTSUP	The value of the schedpolicy or schedparam attributes are not supported.
EPERM	The target thread has insufficient permission to perform the operation or is already engaged in a mutex protocol.
ESRCH	The thread *thread* does not exist.

pthread_setschedprio Subroutine

Purpose

Dynamic thread scheduling parameters access (REALTIME THREADS).

Syntax

```
#include <pthread.h>

int pthread_setschedprio(pthread_t thread, int prio);
```

Description

The **pthread_setschedprio()** function sets the scheduling priority for the thread whose thread ID is given by *thread* to the value given by *prio*. If a thread whose policy or priority has been modified by **pthread_setschedprio()** is a running thread or is runnable, the effect on its position in the tread list depends on the direction of the modification as follows:

- If the priority is raised, the thread becomes the tail of the thread list.
- If the priority is unchanged, the thread does not change position in the thread list.
- If the priority is lowered, the thread becomes the head of the thread list.

Valid priorities are within the range returned by the **sched_get_priority_max()** and **sched_get_priority_min()**.

If the **pthread_setschedprio()** function fails, the scheduling priority of the target thread remains unchanged.

Rationale

The **pthread_setschedprio()** function provides a way for an application to temporarily raise its priority and then lower it again, without having the undesired side-effect of yielding to other threads of the same priority. This is necessary if the application is to implement its own strategies for bounding priority inversion, such as priority inheritance or priority ceilings. This capability is especially important if the implementation does not support the **Thread Priority Protection** or **Thread Priority Inheritance** options; but even if those options are supported, this capability is needed if the application is to bound priority inheritance for other resources, such as semaphores.

The standard developers considered that, while it might be preferable conceptually to solve this problem by modifying the specification of **pthread_setschedparam()**, it was too late to make such a change, because there might be implementations that would need to be changed. Therefore, this new function was introduced.

Return Values

If successful, the **pthread_setschedprio()** function returns 0; otherwise, an error number is returned to indicate the error.

Error Codes

The **pthread_setschedprio()** function might fail if:

Item	Description
EINVAL	The value of *prio* is invalid for the scheduling policy of the specified thread.
ENOTSUP	An attempt was made to set the priority to an unsupported value.
EPERM	The caller does not have the appropriate permission to set the scheduling policy of the specified thread.
EPERM	The implementation does not allow the application to modify the priority to the value specified.
ESRCH	The value specified by *thread* does not refer to an existing thread.

The **pthread_setschedprio** function does not return an error code of [EINTR].

pthread_sigmask Subroutine

Purpose

Examines and changes blocked signals.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <signal.h>

int pthread_sigmask (how, set, oset)
int how;
const sigset_t *set;
sigset_t *oset;
```

Description

Refer to **sigthreadmask**.

pthread_signal_to_cancel_np Subroutine

Purpose

Cancels the specified thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
int pthread_signal_to_cancel_np ( sigset,  thread)
sigset_t *sigset;
pthread_t *thread;
```

Description

The **pthread_signal_to_cancel_np** subroutine cancels the target thread *thread* by creating a handler thread. The handler thread calls the **sigwait** subroutine with the *sigset* parameter, and cancels the target thread when the **sigwait** subroutine returns. Successive calls to this subroutine override the previous ones.

Note:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_signal_to_cancel_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

Parameters

Item	Description
sigset	Specifies the set of signals to wait on.
thread	Specifies the thread to cancel.

Return Values

Upon successful completion, 0 is returned. Otherwise, an error code is returned.

Error Codes

The **pthread_signal_to_cancel_np** subroutine is unsuccessful if the following is true:

Item	Description
EAGAIN	The handler thread cannot be created.
EINVAL	The *sigset* or *thread* parameters are not valid.

pthread_spin_destroy or pthread_spin_init Subroutine

Purpose

Destroys or initializes a spin lock object.

Syntax

```
#include <pthread.h>

int pthread_spin_destroy(pthread_spinlock_t *lock);
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

Description

The **pthread_spin_destroy** subroutine destroys the spin lock referenced by lock and releases any resources used by the lock. The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to the **pthread_spin_init** subroutine. The results are undefined if the **pthread_spin_destroy** subroutine is called when a thread holds the lock, or if this function is called with an uninitialized thread spin lock.

The **pthread_spin_init** subroutine allocates any resources required to use the spin lock referenced by lock and initializes the lock to an unlocked state.

If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is PTHREAD_PROCESS_SHARED, the implementation shall permit the spin lock to be operated upon by any thread that has access to the memory where the spin lock is allocated, even if it is allocated in memory that is shared by multiple processes.

If the Thread Process-Shared Synchronization option is supported and the value of *pshared* is PTHREAD_PROCESS_PRIVATE, or if the option is not supported, the spin lock shall only be operated upon by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined.

The results are undefined if the **pthread_spin_init** subroutine is called specifying an already initialized spin lock. The results are undefined if a spin lock is used without first being initialized.

If the **pthread_spin_init** subroutine function fails, the lock is not initialized and the contents of lock are undefined.

Only the object referenced by lock may be used for performing synchronization.

The result of referring to copies of that object in calls to the **pthread_spin_destroy** subroutine, **pthread_spin_lock** subroutine, **pthread_spin_trylock** subroutine, or the **pthread_spin_unlock** subroutine is undefined.

Return Values

Upon successful completion, these functions shall return zero; otherwise, an error number shall be returned to indicate the error.

Error Codes

Item	Description
EBUSY	The implementation has detected an attempt to initialize or destroy a spin lock while it is in use (for example, while being used in a **pthread_spin_lock** call) by another thread.
EINVAL	The value specified by the *lock* parameter is invalid.

The **pthread_spin_init**subroutine will fail if:

Item	Description
EAGAIN	The system lacks the necessary resources to initialize another spin lock.
ENOMEM	Insufficient memory exists to initialize the lock.

pthread_spin_lock or pthread_spin_trylock Subroutine

Purpose

Locks a spin lock object.

Syntax

```
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
```

Description

The **pthread_spin_lock** subroutine locks the spin lock referenced by the *lock* parameter. The calling thread shall acquire the lock if it is not held by another thread. Otherwise, the thread spins (that is, does not return from the **pthread_spin_lock** call) until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made. The **pthread_spin_trylock** subroutine locks the spin lock referenced by the *lock* parameter if it is not held by any thread. Otherwise, the function fails.

The results are undefined if any of these subroutines is called with an uninitialized spin lock.

Return Values

Upon successful completion, these functions return zero; otherwise, an error number is returned to indicate the error.

Error Codes

Item	Description
EINVAL	The value specified by the *lock* parameter does not refer to an initialized spin lock object.

The **pthread_spin_lock** subroutine fails if:

Item	Description
EDEADLK	The calling thread already holds the lock.

The **pthread_spin_trylock** subroutine fails if:

Item	Description
EBUSY	A thread currently holds the lock.

pthread_spin_unlock Subroutine

Purpose

Unlocks a spin lock object.

Syntax

```
#include <pthread.h>

int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Description

The **pthread_spin_unlock** subroutine releases the spin lock referenced by the *lock* parameter which was locked using the **pthread_spin_lock** subroutine or the **pthread_spin_trylock** subroutine. The results are undefined if the lock is not held by the calling thread. If there are threads spinning on the lock when the **pthread_spin_unlock** subroutine is called, the lock becomes available and an unspecified spinning thread shall acquire the lock.

The results are undefined if this subroutine is called with an uninitialized thread spin lock.

Return Values

Upon successful completion, the **pthread_spin_unlock** subroutine returns zero; otherwise, an error number is returned to indicate the error.

Error Codes

Item	Description
EINVAL	An invalid argument was specified.
EPERM	The calling thread does not hold the lock.

pthread_suspend_np, pthread_unsuspend_np and pthread_continue_np Subroutine

Purpose

Suspends and resume execution of the pthread specified by *thread*.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>

pthread_t thread;
int pthread_suspend_np(thread)
int pthread_unsuspend_np (thread);
int pthread_continue_np(thread);
```

Description

The **pthread_suspend_np** subroutine immediately suspends the execution of the pthread specified by *thread*. On successful return from **pthread_suspend_np**, the suspended pthread is no longer executing. If **pthread_suspend_np** is called for a pthread that is already suspended, the pthread is unchanged and **pthread_suspend_np** returns successful.

Deadlock can occur if **pthread_suspend_np** is used with the following pthread functions.

pthread_getrusage_np
pthread_cancel

pthread_detach
pthread_join
pthread_getunique_np
pthread_join_np
pthread_setschedparam
pthread_getschedparam
pthread_kill

To prevent deadlock, PTHREAD_SUSPENDIBLE=ON should be set.

The `pthread_unsuspend_np` routine decrements the suspend count and once the count is zero, the routine resumes the execution of a suspended pthread. If `pthread_unsuspend_np` is called for a pthread that is not suspended, the pthread is unchanged and `pthread_unsuspend_np` returns successful.

The `pthread_continue_np` routine clears the suspend count and resumes the execution of a suspended pthread. If `pthread_continue_np` is called for a pthread that is not suspended, the pthread is unchanged and `pthread_continue_np` returns successful.

A suspended pthread will not be awakened by a signal. The signal stays pending until the execution of pthread is resumed by **pthread_continue_np**.

Note: Using **pthread_suspend_np** should only be used by advanced users because improper use of this subcommand can lead to application deadlock or the target thread may be suspended holding application locks.

Parameters

Item	Description
thread	Specifies the target thread.

Return Values

Zero is returned when successful. A nonzero value indicates an error.

Error Codes

If any of the following conditions occur, **pthread_suspend_np**, `pthread_unsuspend_np` and **pthread_continue_np** fail and return the corresponding value:

Item	Description
ESRCH	The target thread specified by *thread* attribute cannot be found in the current process.

pthread_unlock_global_np Subroutine

Purpose

Unlocks the global mutex.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
void pthread_unlock_global_np ()
```

Description

The **pthread_unlock_global_np** subroutine unlocks the global mutex when each call to the **pthread_lock_global_np** subroutine is matched by a call to this routine. For example, if a thread called the **pthread_lock_global_np** three times, the global mutex is unlocked after the third call to the **pthread_unlock_global_np** subroutine.

If no threads are waiting for the global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, exactly one thread returns from its call to the **pthread_lock_global_np** subroutine.

Note:

1. The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.
2. The **pthread_unlock_global_np** subroutine is not portable.

This subroutine is not POSIX compliant and is provided only for compatibility with DCE threads. It should not be used when writing new applications.

pthread_yield Subroutine

Purpose

Forces the calling thread to relinquish use of its processor.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
```

```
void pthread_yield ()
```

Description

The **pthread_yield** subroutine forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again. If the run queue is empty when the **pthread_yield** subroutine is called, the calling thread is immediately rescheduled.

If the thread has global contention scope (**PTHREAD_SCOPE_SYSTEM**), calling this subroutine acts like calling the **yield** subroutine. Otherwise, another local contention scope thread is scheduled.

> The **pthread.h** header file must be the first included file of each source file using the threads library. Otherwise, the **-D_THREAD_SAFE** compilation flag should be used, or the cc_r compiler used. In this case, the flag is automatically set.

ptrace, ptracex, ptrace64 Subroutine

Purpose

Traces the execution of another process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/reg.h>
#include <sys/ptrace.h>
#include <sys/ldr.h>
int ptrace ( Request, Identifier, Address, Data, Buffer)
int Request;
int Identifier;
int *Address;
int Data;
int *Buffer;

int ptracex ( request, identifier, addr, data, buff)
int request;
int identifier;
long long addr;
int data;
int *buff;

int ptrace64 ( request, identifier, addr, data, buff)
int request;
long long identifier;
long long addr;
int data;
int *buff;
```

Description

The **ptrace** subroutine allows a 32-bit process to trace the execution of another process. The **ptrace** subroutine is used to implement breakpoint debugging.

A debugged process runs normally until it encounters a signal. Then it enters a stopped state and its debugging process is notified with the **wait** subroutine.

> **Exception:** If the process encounters the **SIGTRAP** signal, a signal handler for **SIGTRAP** exists, and fast traps (Fast Trap Instructions) have been enabled for the process, then the signal handler is called and the debugger is not notified.

While the process is in the stopped state, the debugger examines and modifies the memory image of the process being debugged by using the **ptrace** subroutine. For multi-threaded processes, the **getthrds** subroutine identifies each kernel thread in the debugged process. Also, the debugging process can cause the debugged process to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

As a security measure, the **ptrace** subroutine inhibits the set-user-ID facility on subsequent **exec** subroutines.

When a process running under **ptrace** control calls **load** or **unload**, the debugger is notified and the **W_SLWTED** flag is set in the status returned by **wait**. (A 32-bit process calling **loadbind** is stopped as well.) If the process being debugged has added modules in the shared library to its address space, the modules are added to the process's private copy of the shared library segments. If shared library modules are removed from a process's address space, the modules are deleted from the process's private copy of the library text segment by freeing the pages that contain the module. No other changes to the segment are made, and existing breakpoints do not have to be reinserted.

To allow a debugger to generate code more easily (in order to handle fast trap instructions, for example), memory from the end of the main program up to the next segment boundary can be modified. That memory is read-only to the process but can be modified by the debugger.

When a process being traced forks, the child process is initialized with the unmodified main program and shared library segment, effectively removing breakpoints in these segments in the child process. If multiprocess debugging is enabled, new copies of the main program and shared library segments are made. Modifications to privately loaded modules, however, are not affected by a fork. These breakpoints will remain in the child process, and if these breakpoints are run, a **SIGTRAP** signal is generated and delivered to the process.

If a traced process initiates an **exec** subroutine, the process stops before executing the first instruction of the new image and returns the **SIGTRAP** signal.

Note: The **ptrace** and **ptracex** subroutines are not supported in 64-bit mode.

Fast Trap Instructions

Sometimes, allowing the process being debugged to handle certain trap instructions is useful, instead of causing the process to stop and notify the debugger. You can use this capability to patch running programs or programs whose source codes are not available. For a process to use this capability, you must enable fast traps, which requires you to make a **ptrace** call from a debugger on behalf of the process.

To let a process handle fast traps, a debugger uses the **ptrace (PT_SET**, *pid*, 0, **PTFLAG_FAST_TRAP**, 0**)** subroutine call. Cancel this capability with the **ptrace (PT_CLEAR**, *pid*, 0, **PTFLAG_FAST_TRAP**, 0**)** subroutine call. If a process is able to handle fast traps when the debugger detaches, the fast trap capability remains in effect. Consequently, when another debugger attaches to that process, fast trap processing is still enabled. When no debugger is attached to a process, **SIGTRAP** signals are handled in the same manner, regardless of whether fast traps are enabled.

A fast trap instruction is an unconditional *trap immediate* instruction in the form twi 14,r13,0xNXXX. This instruction has the binary form 0x0ddfNXXX, where N is a hex digit >=8 and XXX are any three hex digits. By using different values of 0xNXXX, a debugger can generate different fast trap instructions, allowing a signal handler to quickly determine how to handle the signal. (The fast trap instruction is defined by the macro **_PTRACE_FASTTRAP**. The **_PTRACE_FASTTRAP_MASK** macro can be used to check whether a trap is a fast trap.)

Usually, a fast trap instruction is treated like any other trap instruction. However, if a process has a signal handler for **SIGTRAP**, the signal is not blocked, and the fast trap capability is enabled, then the signal handler is called and the debugger is not notified.

A signal handler can logically AND the trap instruction with **_PTRACE_FASTTRAP_NUM** (0x7FFF) to obtain an integer identifying which trap instruction was run.

Fast data watchpoint

The `ptrace` subroutine supports the ability to enable fast watchpoint trap handling. This is similar to fast trap instruction handling in that when it is enabled. Processes that have a signal handler for `SIGTRAP` will have the handler called when a watchpoint trap is encountered. In the `SIGTRAP` signal handler, the traced process can detect a fast watchpoint trap by checking the `SI_FAST_WATCH` in the _si_flags of the `siginfo_t` that is passed to the handler. The fast watchpoint handling employs trap-after semantics, which means that the store to the watched location is completed before calling the trap handler, so the instruction address pointer in the signal context that is passed to the handler will point to the instruction following the instruction that caused the trap.

Thread-level tracing

The `ptrace` subroutine supports setting breakpoints and watchpoint per-thread for system scope (1:1) threads. With these, the tracing process (debugger) is only notified when the specific thread of interest has encountered a trap. This provides an efficient means for debuggers to trace individual threads of interest since it doesn't have to filter "false hit" notifications. See the `PTT_WATCH`, `PTT_SET_TRAP`, and `PTT_CLEAR_TRAP` request types below for the usage description.

The ptrace programming model remains unchanged with thread-level breakpoints and watchpoints in that the attachment is still done at the process level, and the target process stops and notifies the tracing process upon encountering a trap. The tracing process can detect that the traced process has stopped for a thread-level trap by checking the TTHRDTRAP flag (in ti_flag2) of the stopping thread (the thread with TTRCSIG set in ti_flag). These flags can be checked by calling getthrds64 on the target process.

Other behaviors that are specific to thread-level tracing:

Thread-level breakpoints

- Clear automatically when all threads for which the breakpoint is active have terminated.
- Not supported for multiprocess debugging (PT_MULTI) . They are cleared upon fork and exec.

Thread-level watchpoints

- Newly created threads inherit the process-level watch location.
- Not inherited across fork and exec.

For the 64-bit Process

Use **ptracex** where the debuggee is a 64-bit process and the operation requested uses the third (*Address*) parameter to reference the debuggee's address space or is sensitive to register size. Note that **ptracex** and **ptrace64** will also support 32-bit debugees.

If returning or passing an **int** doesn't work for a 64-bit debuggee (for example, **PT_READ_GPR**), the buffer parameter takes the address for the result. Thus, with the **ptracex** subroutine, **PT_READ_GPR** and **PT_WRITE_GPR** take a pointer to an 8 byte area representing the register value.

In general, **ptracex** supports all the calls that **ptrace** does when they are modified for any that are extended for 64-bit addresses (for example, GPRs, LR, CTR, IAR, and MSR). Anything whose size increases for 64-bit processes must be allowed for in the obvious way (for example, **PT_REGSET** must be an array of long longs for a 64-bit debuggee).

Parameters

Request
Determines the action to be taken by the **ptrace** subroutine and has one of the following values:

PT_ATTACH
This request allows a debugging process to attach a current process and place it into trace mode for debugging. This request cannot be used if the target process is already being traced. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Address*, *Data*, and *Buffer* parameters are ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to one the following codes:

ESRCH
Process ID is not valid; the traced process is a kernel process; the process is currently being traced; or, the debugger or traced process already exists.

EPERM
Real or effective user ID of the debugger does not match that of the traced process, or the debugger does not have root authority.

EINVAL
The debugger and the traced process are the same.

PT_CLEAR
This request clears an internal flag or capability. The *Data* parameter specifies which flags to clear. The following flag can be cleared:

PTFLAG_FAST_TRAP
Disables the special handling of a fast trap instruction (Fast Trap Instructions). This allows all fast trap instructions causing an interrupt to generate a **SIGTRAP** signal.

The *Identifier* parameter specifies the process ID of the traced process. The *Address* parameter, *Buffer* parameter, and the unused bits in the *Data* parameter are reserved for future use and should be set to 0.

PTFLAG_FAST_WATCH
Enables fast watchpoint trap handling. When a watchpoint trap occurs in a process that has a signal handler for SIGTRAP, and the process has fast watchpoints enabled, the signal handler will be called instead of notifying the tracing process.

PTT_CLEAR_TRAP
This request type clears thread-level breakpoints.

The *Identifier* parameter is a valid kernel thread ID in the target process (-1 for all). The *Address* parameter is the address of the breakpoint. The *Data* parameter must be 0. The *Buffer* parameter must be NULL.

If the request is unsuccessful, -1 is returned and the `errno` global variable is set to one of the following:

ESRCH
The *Identifier* parameter does not refer to a valid kernel thread in the target process, or no breakpoint was found for the target thread at the given *Address*.

EINVAL
The *Data* parameter was non-zero or *Buffer* was non-NULL.

PT_CONTINUE
This request allows the process to resume execution. If the *Data* parameter is 0, all pending signals, including the one that caused the process to stop, are concealed before the process resumes execution. If the *Data* parameter is a valid signal number, the process resumes execution as if it had received that signal. If the *Address* parameter equals 1, the execution continues from where it stopped. If the *Address* parameter is not 1, it is assumed to be the address at which the process should resume execution. Upon successful completion, the value of the *Data* parameter is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
The signal to be sent to the traced process is not a valid signal number.

Note: For the **PT_CONTINUE** request, use **ptracex** or **prtrace64** with a 64-bit debuggee because the resume address needs 64 bits.

PTT_CONTINUE
This request asks the scheduler to resume execution of the kernel thread specified by *Identifier*. This kernel thread must be the one that caused the exception. The *Data* parameter specifies how to handle signals:

- If the *Data* parameter is 0, the kernel thread which caused the exception will be resumed as if the signal never occurred.
- If the *Data* parameter is a valid signal number, the kernel thread which caused the exception will be resumed as if it had received that signal.

The *Address* parameter specifies where to resume execution:

- If the *Address* parameter is 1, execution resumes from the address where it stopped.
- If the *Address* parameter contains an address value other than 1, execution resumes from that address.

The *Buffer* parameter should point to a PTTHREADS structure, which contains a list of kernel thread identifiers to be started. This list should be NULL terminated if it is smaller than the maximum allowed.

On successful completion, the value of the *Data* parameter is returned to the debugging process. On unsuccessful completion, the value -1 is returned, and the **errno** global variable is set as follows:

EINVAL
The *Identifier* parameter names the wrong kernel thread.

EIO
The signal to be sent to the traced kernel thread is not a valid signal number.

ESRCH
The *Buffer* parameter names an invalid kernel thread. Each kernel thread in the list must be stopped and belong to the same process as the kernel thread named by the *Identifier* parameter.

Note: For the **PTT_CONTINUE** request, use **ptracex** or **ptrace64** with a 64-bit debuggee because the resume address needs 64 bits.

PT_DETACH
This request allows a debugged process, specified by the *Identifier* parameter, to exit trace mode. The process then continues running, as if it had received the signal whose number is contained in the *Data* parameter. The process is no longer traced and does not process any further **ptrace** calls. The *Address* and *Buffer* parameters are ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
Signal to be sent to the traced process is not a valid signal number.

PT_GET_UKEY
This request reads the user-key assigned to a specific effective address indicated by the *address* parameter into the location pointed to the *buffer* parameter. The process ID of the traced process must be passed in the *identifier* parameter. The *data* parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

ENOSYS
Process is not user-key aware.

PT_KILL
This request allows the process to terminate the same way it would with an **exit** subroutine.

PT_LDINFO

This request retrieves a description of the object modules that were loaded by the debugged process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored. The *Address* parameter specifies the location where the loader information is copied. The *Data* parameter specifies the size of this area. The loader information is retrieved as a linked list of **ld_info** structures. The first element of the list corresponds to the main executable module. The **ld_info** structures are defined in the **/usr/include/sys/ldr.h** file. The linked list is

implemented so that the `ldinfo_next` field of each element gives the offset of the next element from this element. The `ldinfo_next` field of the last element has the value 0.

Each object module reported is opened on behalf of the debugger process. The file descriptor for an object module is saved in the `ldinfo_fd` field of the corresponding **ld_info** structure. The debugger process is responsible for managing the files opened by the **ptrace** subroutine.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

ENOMEM
Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.

Note: For the **PT_LDINFO** request, use **ptracex** or **ptrace64** with a 64-bit debuggee because the source address needs 64 bits.

PT_LDXINFO

This request is similar to the **PT_LDINFO** request. A linked list of **ld_xinfo** structures is returned instead of a list of **ld_info** structures. The first element of the list corresponds to the main executable module. The **ld_xinfo** structures are defined in the **/usr/include/sys/ldr.h** file. The linked list is implemented so that the `ldinfo_next` field of each element gives the offset of the next element from this element. The `ldinfo_next` field of the last element has the value 0.

Each object module reported is opened on behalf of the debugger process. The file descriptor for an object module is saved in the `ldinfo_fd` field of the corresponding **ld_xinfo** structure. The debugger process is responsible for managing the files opened by the **ptrace** subroutine.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

ENOMEM
Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.

Note: For the **PT_LDXINFO** request, use **ptracex** or **ptrace64** with a 64-bit debuggee because the source address needs 64 bits.

PT_MULTI
This request turns multiprocess debugging mode on and off, to allow debugging to continue across **fork** and **exec** subroutines. A 0 value for the *Data* parameter turns multiprocess debugging mode off, while all other values turn it on. When multiprocess debugging mode is in effect, any **fork** subroutine allows both the traced process and its newly created process to trap on the next instruction. If a traced process initiated an **exec** subroutine, the process stops before executing the first instruction of the new image and returns the **SIGTRAP** signal. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Address* and *Buffer* parameters are ignored.

Also, when multiprocess debugging mode is enabled, the following values are returned from the **wait** subroutine:

W_SEWTED
Process stopped during execution of the **exec** subroutine.

W_SFWTED
Process stopped during execution of the **fork** subroutine.

PT_READ_BLOCK
This request reads a block of data from the debugged process address space. The *Address* parameter points to the block of data in the process address space, and the *Data* parameter gives its length in bytes. The value of the *Data* parameter must not be greater than 1024. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter points to

the location in the debugging process address space where the data is copied. Upon successful completion, the **ptrace** subroutine returns the value of the *Data* parameter.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to one of the following codes:

EIO
The *Data* parameter is less than 1 or greater than 1024.

EIO
The *Address* parameter is not a valid pointer into the debugged process address space.

EFAULT
The *Buffer* parameter does not point to a writable location in the debugging process address space.

Note: For the **PT_READ_BLOCK** request, use **ptracex** or **ptrace64** with a 64-bit debuggee because the source address needs 64 bits.

PT_READ_FPR
This request stores the value of a floating-point register into the location pointed to by the *Address* parameter. The *Data* parameter specifies the floating-point register, defined in the **sys/reg.h** file for the machine type on which the process is run. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
The *Data* parameter is not a valid floating-point register. The *Data* parameter must be in the range 256-287.

PTT_READ_FPRS
This request writes the contents of the 32 floating point registers to the area specified by the *Address* parameter. This area must be at least 256 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

PTT_READ_FPSCR_HI
This request writes the contents of the upper 32-bits of the FPSCR register to the area specified by the *Address* parameter. This area must be at least 4 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

PTT_READ_TM
This request reads the Transactional Memory (TM) state of the specified thread. The data format is a `__tm_context_t` structure that contains the TM Special Purpose Registers (SPRs) (TEXASR, TFIAR, and TFHAR) and the checkpoint state, including all of the problem-state writable registers with the exception of CR0, FXCC, EBBHR, EBBRR, BESCR, and the performance monitor registers.

PTT_WRITE_FPSCR_HI
This request updates the contents of the upper 32-bits of the FPSCR register with the value specified in the area designated by the *Address* parameter. This area must be at least 4 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

PT_READ_GPR
This request returns the contents of one of the general-purpose or special-purpose registers of the debugged process. The *Address* parameter specifies the register whose value is returned. The value of the *Address* parameter is defined in the **sys/reg.h** file for the machine type on which the process is run. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Data* and *Buffer* parameters are ignored. The buffer points to long long target area.

Note: If **ptracex** or **ptrace64** with a 64-bit debuggee is used for this request, the register value is instead returned to the 8-byte area pointed to by the buffer pointer.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
The *Address* is not a valid general-purpose or special-purpose register. The *Address* parameter must be in the range 0-31 or 128-136.

PTT_READ_GPRS
This request writes the contents of the 32 general purpose registers to the area specified by the *Address* parameter. This area must be at least 128 bytes long.

Note: If **ptracex** or **ptrace64** are used with a 64-bit debuggee for the **PTT_READ_GPRS** request, there must be at least a 256 byte target area. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

PT_READ_I or PT_READ_D
These requests return the word-aligned address in the debugged process address space specified by the *Address* parameter. On all machines currently supported by AIX Version 4, the **PT_READ_I** and **PT_READ_D** instruction and data requests can be used with equal results. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Data* parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
The *Address* is not word-aligned, or the *Address* is not valid. User blocks, kernel segments, and kernel extension segments are not considered as valid addresses.

Note: For the **PT_READ_I** or the **PT_READ_D** request, use **ptracex** or **ptrace64** with a 64-bit debuggee because the source address needs 64 bits.

PTT_READ_SPRS
This request writes the contents of the special purpose registers to the area specified by the *Address* parameter, which points to a **ptsprs** structure. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

Note: For the **PTT_READ_SPRS** request, use **ptracex** or **ptrace64** with the 64-bit debuggee because the new **ptxsprs** structure must be used.

PTT_READ_UKEYSET
This request reads the active user-key-set for the specified thread whose thread ID is specified by the *identifier* parameter into the location pointed to the *buffer* parameter. The *address* and *data* parameters are ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

ENOSYS
Process is not user-key aware.

PTT_READ_VEC
This request reads the vector register state of the specified thread. The data format is a `__vmx_context_t` structure that contains the 32 vector registers, in addition to the VSCR and VRSAVE registers.

PT_REATT
This request allows a new debugger, with the proper permissions, to trace a process that was already traced by another debugger. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Address*, *Data*, and *Buffer* parameters are ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to one the following codes:

ESRCH
The *Identifier* is not valid; or the traced process is a kernel process.

EPERM
Real or effective user ID of the debugger does not match that of the traced process, or the debugger does not have root authority.

EINVAL
The debugger and the traced process are the same.

PT_REGSET
This request writes the contents of all 32 general purpose registers to the area specified by the *Address* parameter. This area must be at least 128 bytes for the 32-bit debuggee or 256 bytes for the 64-bit debuggee. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Data* and *Buffer* parameters are ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
The *Address* parameter points to a location outside of the allocated address space of the process.

Note: For the **PT_REGSET** request, use **ptracex** or **ptrace64** with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned.

PT_SET
This request sets an internal flag or capability. The *Data* parameter indicates which flags are set. The following flag can be set:

PTFLAG_FAST_TRAP
Enables the special handling of a fast trap instruction (Fast Trap Instructions). When a fast trap instruction is run in a process that has a signal handler for **SIGTRAP**, the signal handler will be called even if the process is being traced.

The *Identifier* parameter specifies the process ID of the traced process. The *Address* parameter, *Buffer* parameter, and the unused bits in the *Data* parameter are reserved for future use and should be set to 0.

PTT_SET_TRAP
This request type sets thread-level breakpoints.

The *Identifier* parameter is a valid kernel ID in the target process. The *Address* parameter is the address in the target process for the breakpoint. The *Data* parameter is the length of data in *Buffer*, it must be 4. The *Buffer* parameter is a pointer to trap instruction to be written.

The system call will not evaluate the contents of the buffer for this request, but by convention, it should contain a single trap instruction.

If the request is unsuccessful, a value of -1 is returned and the `errno` global variable is set to one of the following:

ENOMEM
Could not allocate kernel memory.

ESRCH
The *Identifier* parameter does not refer to a valid kernel thread in the target process.

EIO
The *Address* parameter does not point to a writable location in the address space of the target process.

EINVAL
Data parameter was not 4, or the target thread already has a breakpoint set at *Address*.

EFAULT
The *Buffer* parameter does not point to a readable location in the caller's address space.

PT_TRACE_ME
This request must be issued by the debugged process to be traced. Upon receipt of a signal, this request sets the process trace flag, placing the process in a stopped state, rather than the action specified by the **sigaction** subroutine. The *Identifier*, *Address*, *Data*, and *Buffer* parameters are ignored. Do not issue this request if the parent process does not expect to trace the debugged process.

As a security measure, the **ptrace** subroutine inhibits the set-user-ID facility on subsequent **exec** subroutines, as shown in the following example:

```
if((childpid = fork()) == 0)
{ /* child process */
 ptrace(PT_TRACE_ME,0,0,0,0);
   execlp(          )/* your favorite exec*/
   }
else
{    /* parent                          */
     /* wait for child to stop          */
        rc = wait(status)
```

Note: This is the only request that should be performed by the child. The parent should perform all other requests when the child is in a stopped state.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

ESRCH
Process is debugged by a process that is not its parent.

PT_WATCH
This request allows to have a watchpoint on the memory region specified when the debugged process changes the content at the specified memory region.

The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored. The *Address* parameter specifies beginning of the memory region to be watched. To clear the watchpoint the *Address* parameter must be NULL. The *Data* parameter specifies the size of the memory region.

Watchpoints are supported only on the hardware POWER630, POWER5 and POWER6. Currently the size of the memory region, that is, the parameter *Data* must be 8 because only 8 byte watchpoint is supported at the hardware level.

If this request is unsuccessful, a value of -1 is returned and the `errno` global variable is set to the following code:

EPERM
If the hardware does not support watchpoints or if specified *Identifier* is not valid Process ID.

EIO
If the specified *Address* is not double word aligned.

EINVAL
If the specified *Data* is not 8.

PTT_WATCH
This request sets and clears thread-level watchpoints.

The *Identifier* parameter is a valid kernel thread ID in the target process (-1 for all). The *Address* parameter is the double-worded aligned address to watch. A value of 0 clears the watchpoint. The *Data* parameter must be 0 (clear) or 8 (set). The *Buffer* parameter must be NULL.

If the request is unsuccessful, a value of -1 is returned and the `errno` global variable is set to one of the following:

ESRCH
The *Identifier* parameter does not refer to a valid kernel thread in the target process.

EPERM
The hardware watchpoint facility is not supported on the platform.

EIO
The requested *Address* is not a valid, double-worded aligned address in target process address space, or the *Address* is non-zero and *Data* is not *8*

PT_WRITE_BLOCK
This request writes a block of data into the debugged process address space. The *Address* parameter points to the location in the process address space to be written into. The *Data* parameter gives the length of the block in bytes, and must not be greater than 1024. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter points to the location in the debugging process address space where the data is copied. Upon successful completion, the value of the *Data* parameter is returned to the debugging process.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to one of the following codes:

EIO
The *Data* parameter is less than 1 or greater than 1024.

EIO
The *Address* parameter is not a valid pointer into the debugged process address space.

EFAULT
The *Buffer* parameter does not point to a readable location in the debugging process address space.

Note: For the **PT_WRITE_BLOCK** request, use **ptracex** or **ptrace64** with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned.

PT_WRITE_FPR
This request sets the floating-point register specified by the *Data* parameter to the value specified by the *Address* parameter. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
The *Data* parameter is not a valid floating-point register. The *Data* parameter must be in the range 256-287.

PTT_WRITE_FPRS
This request updates the contents of the 32 floating point registers with the values specified in the area designated by the *Address* parameter. This area must be at least 256 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

PT_WRITE_GPR
This request stores the value of the *Data* parameter in one of the process general-purpose or special-purpose registers. The *Address* parameter specifies the register to be modified. Upon

successful completion, the value of the *Data* parameter is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

Note: If **ptracex** or **ptrace64** are used with a 64-bit debuggee for the **PT_WRITE_GPR** request, the new register value is NOT passed via the *Data* parameter, but is instead passed via the 8-byte area pointed to by the buffer parameter.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
The *Address* parameter is not a valid general-purpose or special-purpose register. The *Address* parameter must be in the range 0-31 or 128-136.

PTT_WRITE_GPRS
This request updates the contents of the 32 general purpose registers with the values specified in the area designated by the *Address* parameter. This area must be at least 128 bytes long. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

Note: For the **PTT_WRITE_GPRS** request, use **ptracex** or **ptrace64** with the 64-bit debuggee because 64-bit registers requiring 256 bytes are returned. The buffer points to long long source area.

PT_WRITE_I or PT_WRITE_D
These requests write the value of the *Data* parameter into the address space of the debugged process at the word-aligned address specified by the *Address* parameter. On all machines currently supported by AIX Version 4, instruction and data address spaces are not separated. The **PT_WRITE_I** and **PT_WRITE_D** instruction and data requests can be used with equal results. Upon successful completion, the value written into the address space of the debugged process is returned to the debugging process. The *Identifier* parameter is interpreted as the process ID of the traced process. The *Buffer* parameter is ignored.

If this request is unsuccessful, a value of -1 is returned and the **errno** global variable is set to the following code:

EIO
The *Address* parameter points to a location in a pure procedure space and a copy cannot be made; the *Address* is not word-aligned; or, the *Address* is not valid. User blocks, kernel segments, and kernel extension segments are not considered valid addresses.

Note: For the or **PT_WRITE_I** or **PT_WRITE_D** request, use **ptracex** or **ptrace64** with a 64-bit debuggee because the target address needs 64 bits.

PTT_WRITE_SPRS
This request updates the special purpose registers with the values in the area specified by the *Address* parameter, which points to a **ptsprs** structure. The *Identifier* parameter specifies the traced kernel thread. The *Data* and *Buffer* parameters are ignored.

Identifier
Determined by the value of the *Request* parameter.

Address
Determined by the value of the *Request* parameter.

Data
Determined by the value of the *Request* parameter.

Buffer
Determined by the value of the *Request* parameter.

Note: For the **PTT_READ_SPRS** request, use **ptracex** or **ptrace64** with the 64-bit debuggee because the new **ptxsprs** structure must be used.

PTT_WRITE_VEC
This request writes the vector register state of the specified thread. The data format is a `__vmx_context_t` structure that contains the 32 vector registers, in addition to the VSCR and VRSAVE registers.

Error Codes

The **ptrace** subroutine is unsuccessful when one of the following is true:

Item	Description
EFAULT	The *Buffer* parameter points to a location outside the debugging process address space.
EINVAL	The debugger and the traced process are the same; or the *Identifier* parameter does not identify the thread that caused the exception.
EIO	The *Request* parameter is not one of the values listed, or the *Request* parameter is not valid for the machine type on which the process is run.
ENOMEM	Either the area is not large enough to accommodate the loader information, or there is not enough memory to allocate an equivalent buffer in the kernel.
ENXIO	The target thread has not referenced the VMX unit and is not currently a VMX thread.
EPERM	The *Identifier* parameter corresponds to a kernel thread which is stopped in kernel mode and whose computational state cannot be read or written.
ESRCH	The *Identifier* parameter identifies a process or thread that does not exist, that has not run a **ptrace** call with the **PT_TRACE_ME** request, or that is not stopped.

For **ptrace**: If the debuggee is a 64-bit process, the options that refer to GPRs or SPRs fail with errno = **EIO**, and the options that specify addresses are limited to 32-bits.

For **ptracex** or **ptrace64**: If the debuggee is a 32-bit process, the options that refer to GPRs or SPRs fail with errno = **EIO**, and the options that specify addresses in the debuggee's address space that are larger than 2**32 - 1 fail with errno set to **EIO**.

Also, the options **PT_READ_U** and **PT_WRITE_U** are not supported if the debuggee is a 64-bit program (errno = **ENOTSUP**).

ptsname Subroutine

Purpose

Returns the name of a pseudo-terminal device.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

char *ptsname (*FileDescriptor*)
int *FileDescriptor*

Description

The **ptsname** subroutine gets the path name of the worker pseudo-terminal associated with the controller pseudo-terminal device defined by the *FileDescriptor* parameter.

Parameters

Item	Description
FileDescriptor	Specifies the file descriptor of the controller pseudo-terminal device

Return Values

The **ptsname** subroutine returns a pointer to a string containing the null-terminated path name of the pseudo-terminal device associated with the file descriptor specified by the *FileDescriptor* parameter. A null pointer is returned and the **errno** global variable is set to indicate the error if the file descriptor does not describe a pseudo-terminal device in the **/dev** directory.

Files

Item	Description
/dev/*	Terminal device special files.

putauthattr Subroutine

Purpose

Modifies the authorizations that are defined in the authorization database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putauthattr(Auth, Attribute, Value, Type)
    char *Auth;
    char *Attribute;
    void *Value;
    int Type;
```

Description

The **putauthattr** subroutine modifies the authorization database. The subroutine can be invoked only by new authorizations or authorizations that already exist in the user-defined authorization database. Calling the **putauthattr** subroutine with an authorization in the system-defined authorization table will fail.

New authorizations can be added to the authorization database by calling the **putauthattr** subroutine with the **SEC_NEW** type and specifying the new authorization name. Authorization names are of a hierarchical structure (that is, parent.subparent.subsubparent). Parent authorizations must exist before the child can be created. Deletion of an authorization or authorization attribute is done using the **SEC_DELETE** type for the **putauthattr** subroutine. Deleting an authorization requires that all child authorizations have already been deleted.

Data changed by the **putauthattr** subroutine must be explicitly committed by calling the **putauthattr** subroutine with a *Type* parameter specifying the **SEC_COMMIT** type. Until all the data is committed, only the **getauthattr** and **getauthattrs** subroutines within the process return the modified data. Changes that

are made to the authorization database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command or until the system is rebooted.

Parameters

Item	Description
Auth	The authorization name. This parameter must be specified unless the *Type* parameter is **SEC_COMMIT**.
Attribute	Specifies the attribute to be written. The following possible attributes are defined in the **usersec.h** file: **S_DFLTMSG** Specifies a default authorization description to use if message catalogs are not in use. The attribute type is **SEC_CHAR**. **S_ID** Specifies a unique integer that is used to identify the authorization. The attribute type is **SEC_INT**. **Note:** Do not modify this value after it is set initially when the authorization is created. Modifying the value might compromise the security of the system. **S_MSGCAT** Specifies the message catalog file name that contains the description of the authorization. The attribute type is **SEC_CHAR**. **S_MSGSET** Specifies the message set that contains the message for the description of the authorization in the file specified by the **S_MSGCAT** attribute. The attribute type is **SEC_INT**. **S_MSGNUMBER** Specifies the message number for the description of the authorization in the file that is specified by the **S_MSGCAT** attribute and the message set that is specified by the **S_MSGSET** attribute. The attribute type is **SEC_INT**.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer according to the values of the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Item	Description
Type	Specifies the type of attribute. The following valid types are defined in the **usersec.h** file: **SEC_INT** The format of the attribute is an integer. The user should supply an integer value. **SEC_CHAR** The format of the attribute is a null-terminated character string. The user should supply a character pointer. **SEC_LIST** The format of the attribute is a series of concatenated strings, each of which is null-terminated. The last string in the series is terminated by two successive null characters. The user should supply a character pointer. **SEC_COMMIT** Specifies that the changes to the named authorization are to be committed to permanent storage. The values of the *Attribute* and *Value* parameters are ignored. If no authorization is specified, the changes to all modified authorizations are committed to permanent storage. **SEC_DELETE** If the *Attribute* parameter is specified, the corresponding attribute is deleted from the authorization database. If no *Attribute* parameter is specified, the entire authorization definition is deleted from the authorization database. **SEC_NEW** Creates a new authorization in the authorization database.

Security

Files Accessed:

File	Mode
/etc/security/authorizations	rw

Return Values

If successful, the **putauthattr** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **putauthattr** subroutine fails, one of the following **errno** values is set:

Item	Description
EEXIST	The *Type* parameter is **SEC_DELETE** and the *Auth* parameter specifies an authorization that is the parent of at least one another authorization.
EINVAL	The *Auth* parameter is **NULL** and the *Type* parameter is not **SEC_COMMIT**.
EINVAL	The *Auth* parameter is **default, ALL, ALLOW_OWNER, ALLOW_GROUP** or **ALLOW_ALL**.
EINVAL	The *Auth* parameter begins with **aix**. Authorizations with a hierarchy that begin with **aix** are reserved for system-defined authorizations and are not modifiable using the **putauthattr** subroutine.
EINVAL	The *Attribute* parameter is **NULL** and the *Type* parameter is not **SEC_NEW**, **SEC_DELETE** or **SEC_COMMIT**.

Item	Description
EINVAL	The *Attribute* parameter does not contain one of the defined attributes.
EINVAL	The *Type* parameter does not contain one of the defined values.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
ENOENT	The authorization specified by the *Auth* parameter does not exist.
ENOENT	The *Auth* parameter specifies a hierarchy and the *Type* parameter is **SEC_NEW**, but the parent authorization does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.

putauthattrs Subroutine

Purpose

Modifies multiple authorization attributes in the authorization database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putauthattrs(Auth, Attributes, Count)
    char *Auth;
    dbattr_t *Attributes;
    int Count;
```

Description

The **putauthattrs** subroutine modifies one or more attributes from the authorization database. The subroutine can be called only with an authorization that already exists in the user-defined authorization database. Calling the **putauthattrs** subroutine with an authorization in the system-defined authorization table fails.

The **putauthattrs** subroutine is used to modify attributes of existing authorizations only. To create or remove user-defined authorizations, use the **putauthattr** subroutine instead. Data changed by the **putauthattrs** subroutine must be explicitly committed by calling the **putauthattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. When all the data is committed, only the **getauthattr** and **getauthattrs** subroutines within the process return the modified data. Changes that are made to the authorization database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command.

The *Attributes* array contains information about each attribute that is to be updated. Each value specified in the *Attributes* array must be examined on a successful call to the **putauthattrs** subroutine to determine whether the value of the *Attributes* array was successfully written. The **dbattr_t data** structure contains the following fields:

Item	Description
attr_name	The name of the authorization attribute to update.
attr_idx	This attribute is used internally by the **putauthattrs** subroutine.

Item	Description
attr_type	The type of the attribute that is being updated.
attr _flag	The result of the request to update the target attribute. On successful completion, a value of zero is returned. Otherwise, a value of nonzero value is returned.
attr_un	A union that contains the value to update the requested attribute with.
attr_domain	This field is ignored by the **putauthattrs** subroutine.

The following valid authorization attributes for the **putauthattrs** subroutine are defined in the **usersec.h** file:

Name	Description	Type
S_DFLTMSG	The default authorization description that is used when catalogs are not in use.	**SEC_CHAR**
S_ID	A unique integer that is used to identify the authorization. **Note:** After the value is set initially, it must not be modified because it might be in use on the system.	**SEC_INT**
S_MSGCAT	The message catalog name that contains the authorization description.	**SEC_CHAR**
S_MSGSET	The message catalog's set number for the authorization description.	**SEC_INT**
S_MSGNUMBER	The message number for the authorization description.	**SEC_INT**

The following union members correspond to the definitions of the **attr_char**, **attr_int**, **attr_long** and the **attr_llong** macros in the **usersec.h** file respectively.

Item	Description
au_char	A character pointer to the value that is to be written for attributes of **SEC_CHAR** and **SEC_LIST** types.
au_int	Integer value that is to be written for attributes of the **SEC_INT** type.
au_long	Long value that is to be written for attributes of the **SEC_LONG** type.
au_llong	Long long value that is to be written for attributes of the **SEC_LLONG** type.

Parameters

Item	Description
Auth	Specifies the authorization name for which the attributes are to be updated.
Attributes	A pointer to an array of zero or more attributes of the **dbattr_t** type. The list of authorization attributes is defined in the **usersec.h** header file.

Item	Description
Count	The number of array elements in the *Attributes* parameter.

Security

Files Accessed:

File	Mode
/etc/security/ authorizations	rw

Return Values

If the authorization specified by the *Auth* parameter exists in the authorization database, the **putauthattrs** subroutine returns zero, even in the case when no attributes in the *Attributes* array are successfully updated. On successful completion, the **attr_flag** attribute of each value that is specified in the *Attributes* array must be examined to determine whether it was successfully updated. If the specified authorization does not exist, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **putauthattrs** returns -1, one of the following **errno** values is set:

Item	Description
EINVAL	The *Auth* parameter is **NULL, default, ALL, ALLOW_OWNER, ALLOW_GROUP**, or **ALLOW_ALL**.
EINVAL	The *Auth* parameter begins with **aix**. Authorizations with a hierarchy that begin with **aix** are reserved for system-defined authorizations and are not modifiable through the **putauthattrs** subroutine.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Attributes* array is **NULL** and the *Count* parameter is greater than zero.
EINVAL	The *Attributes* array does not point to valid data for the requested attribute.
ENOENT	The authorization specified by the *Auth* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

If the **putauthattrs** subroutine fails to update an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The invoker does not have write access to the authorization database.
EINVAL	The **attr_name** field in the *Attributes* entry is not a recognized authorization attribute.
EINVAL	The **attr_type** field in the *Attributes* entry contains a type that is not valid.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute.

putc, putchar, fputc, or putw Subroutine

Purpose

Writes a character or a word to a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

```
#include <stdio.h>

int putc ( Character,  Stream)
int Character;
FILE *Stream;

int putchar (Character)
int Character;

int fputc (Character, Stream)
int Character;
FILE *Stream;

int putw ( Word, Stream)
int Word;
FILE *Stream;
```

Description

The **putc** and **putchar** macros write a character or word to a stream. The **fputc** and **putw** subroutines serve similar purposes but are true subroutines.

The **putc** macro writes the character *Character* (converted to an **unsigned char** data type) to the output specified by the *Stream* parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

The **putchar** macro is the same as the **putc** macro except that **putchar** writes to the standard output.

The **fputc** subroutine works the same as the **putc** macro, but **fputc** is a true subroutine rather than a macro. It runs more slowly than **putc,** but takes less space per invocation.

Because **putc** is implemented as a macro, it incorrectly treats a *Stream* parameter with side effects, such as **putc(C, *f++)**. For such cases, use the **fputc** subroutine instead. Also, use **fputc** whenever you need to pass a pointer to this subroutine as a parameter to another subroutine.

The **putc** and **putchar** macros have also been implemented as subroutines for ANSI compatibility. To access the subroutines instead of the macros, insert **#undef putc** or **#undef putchar** at the beginning of the source file.

The **putw** subroutine writes the word (**int** data type) specified by the *Word* parameter to the output specified by the *Stream* parameter. The word is written at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. The **putw** subroutine does not assume or cause special alignment of the data in the file.

After the **fputcw**, **putwc**, **fputc**, **putc**, **fputs**, **puts**, or **putw** subroutine runs successfully, and before the next successful completion of a call either to the **fflush** or **fclose** subroutine on the same stream or to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Because of possible differences in word length and byte ordering, files written using the **putw** subroutine are machine-dependent, and may not be readable using the **getw** subroutine on a different processor.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen** subroutine causes it to become buffered or line-buffered. Use the **setbuf** subroutine to change the stream buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested).

Parameters

Item	Description
Stream	Points to the file structure of an open file.
Character	Specifies a character to be written.
Word	Specifies a word to be written (not portable because word length and byte-ordering are machine-dependent).

Return Values

Upon successful completion, these functions each return the value written. If these functions fail, they return the constant **EOF**. They fail if the *Stream* parameter is not open for writing, or if the output file size cannot be increased. Because the **EOF** value is a valid integer, you should use the **ferror** subroutine to detect **putw** errors.

Error Codes

The **fputc** subroutine will fail if either the *Stream* is unbuffered or the *Stream* buffer needs to be flushed, and:

Item	Description
EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying *Stream* and the process would be delayed in the write operation.
EBADF	The file descriptor underlying *Stream* is not a valid file descriptor open for writing.
EFBIG	An attempt was made to write a file that exceeds the file size of the process limit or the maximum file size.
EFBIG	The file is a regular file and an attempt was made to write at or beyond the offset maximum.
EINTR	The write operation was terminated due to the receipt of a signal, and either no data was transferred or the implementation does not report partial transfers for this file. **Note:** Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** Subroutine regarding **sa_restart**.
EIO	A physical I/O error has occurred, or the process is a member of a background process group attempting to perform a **write** subroutine to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring nor blocking the **SIGTTOU** signal and the process group of the process is orphaned. This error may also be returned under implementation-dependent conditions.
ENOSPC	There was no free space remaining on the device containing the file.

Item	Description
EPIPE	An attempt is made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.

The **fputc** subroutine may fail if:

Item	Description
ENOMEM	Insufficient storage space is available.
ENXIO	A request was made of a nonexistent device, or the request was outside the capabilities of the device.

putcmdattr Subroutine

Purpose

Modifies the command security information in the privileged command database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putcmdattr (Command, Attribute, Value, Type)
    char *Command;
    char *Attribute;
    void *Value;
    int Type;
```

Description

The **putcmdattr** subroutine writes a specified attribute into the command database. If the database is not open, this subroutine does an implicit open for reading and writing. Data changed by the **putcmdattr** subroutine must be explicitly committed by calling the **putcmdattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the subroutines within the process return written data.

New entries in the command databases must first be created by invoking the **putcmdattr** subroutine with the **SEC_NEW** type.

Changes that are made to the privileged command database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command or until the system is rebooted.

Parameters

Item	Description
Command	The command name. The value should be the full path to the command on the system. This parameter must be specified unless the *Type* parameter is **SEC_COMMIT**.

Item	Description
Attribute	Specifies the attribute that is to written. The following possible attributes are defined in the **usersec.h** file: **S_ACCESSAUTHS** Access authorizations. The attribute type is **SEC_LIST** and is a null-separated list of authorization names. Sixteen authorizations can be specified. A user with any one of the authorizations can run the command. In addition to the user-defined and system-defined authorizations available on the system, the following three special values can be specified: **ALLOW_OWNER** Allows the command owner to run the command without checking for access authorizations. **ALLOW_GROUP** Allows the command group to run the command without checking for access authorizations. **ALLOW_ALL** Allows every user to run the command without checking for access authorizations. **S_AUTHPRIVS** Authorized privileges. The attribute type is **SEC_LIST**. Privilege authorization and authorized privileges pairs indicate process privileges during the execution of the command corresponding to the authorization that the parent process possesses. The authorization and its corresponding privileges are separated by an equal sign (=); individual privileges are separated by a plus sign (+); the authorization and privileges pairs are separated by a comma (,) as shown in the following illustration: `auth=priv+priv+...,auth=priv+priv...,...` The number of authorization/privileges pairs is limited to sixteen. **S_AUTHROLES** A role or list of roles, users having these roles have to be authenticated to allow execution of the command. The attribute type is **SEC_LIST**. **S_INNATEPRIVS** Innate privileges. This is a null-separated list of privileges assigned to the process when running the command. The attribute type is **SEC_LIST**. **S_INHERITPRIVS** Inheritable privileges. This is a null-separated list of privileges that is passed to child processes privileges. The attribute type is **SEC_LIST**. **S_EUID** The effective user ID to be assumed when running the command. The attribute type is **SEC_INT**. **S_EGID** The effective group ID to be assumed when running the command. The attribute type is **SEC_INT**. **S_RUID** The real user ID to be assumed when running the command. The attribute type is **SEC_INT**.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer according to the values of the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Item	Description
Type	Specifies the type of attribute. The following valid types are defined in the **usersec.h** file: **SEC_INT** The format of the attribute is an integer. **SEC_CHAR** The format of the attribute is a null-terminated character string. The user should supply a character pointer. **SEC_LIST** The format of the attribute is a series of concatenated strings, each of which is null-terminated. The last string in the series is terminated by two successive null characters. For the **putcmdattr** subroutine, the user should supply a character pointer. **SEC_COMMIT** For the **putcmdattr** subroutine, this value specified by itself indicates that changes to the named command are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no command is specified, the changes to all modified commands are committed to permanent storage. **SEC_DELETE** If the *Attribute* parameter is specified, the corresponding attribute is deleted from the privileged command database. If no *Attribute* parameter is specified, the entire command definition is deleted from the privileged command database. **SEC_NEW** Creates a new command in the privileged command database when it is specified with the **putcmdattr** subroutine.

Security

Files Accessed:

File	Mode
/etc/security/privcmds	rw

Return Values

If successful, the **putcmdattr** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **putcmdattr** subroutine fails, one of the following **errno** values can be set:

Item	Description
EINVAL	The *Command* parameter is **NULL** and the *Type* parameter is not **SEC_COMMIT**.
EINVAL	The *Command* parameter is **default** or **ALL**.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or is **NULL**.
EINVAL	The *Type* parameter does not contain one of the defined values.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
ENOENT	The command specified by the *Command* parameter does not exist.

Item	Description
EPERM	The operation is not permitted.

putcmdattrs Subroutine

Purpose

Modifies multiple command attributes in the privileged command database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putcmdattrs(Command, Attributes, Count)
    char *Command;
    dbattr_t *Attributes;
    int Count;
```

Description

The **putcmdattrs** subroutine modifies one or more attributes from the privileged command database. If the database is not open, this subroutine does an implicit open for reading and writing. The command specified by the *Command* parameter must include the full path to the command and exist in the privileged command database.

The **putcmdattrs** subroutine is only used to modify attributes of existing commands in the database. To create or remove command entries, use the **putcmdattr** subroutine instead. Data changed by the **putcmdattrs** subroutine must be explicitly committed by calling the **putcmdattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getcmdattr** and **getcmdattrs** subroutines within the process return the modified data. Changes made to the privileged command database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command or until the system is rebooted.

The *Attributes* parameter contains information about each attribute that is to be updated. Each values that is specified in the *Attributes* parameter must be examined on a successful call to the **putcmdattrs** subroutine to determine whether the *Attributes* parameter was successfully written. The **dbattr_t** data structure contains the following fields:

Name	Description	Type
S_ACCESSAUTHS	Access authorizations, a null-separated list of authorization names. Sixteen authorizations can be specified. A user with any one of the authorizations can run the command. In addition to the user-defined and system-defined authorizations available on the system, the following three special values can be specified: **ALLOW_OWNER** Allows the command owner to run the command without checking for access authorizations. **ALLOW_GROUP** Allows the command group to run the command without checking for access authorizations. **ALLOW_ALL** Allows every user to run the command without checking for access authorizations.	**SEC_LIST**

Name	Description	Type
S_AUTHPRIVS	Authorized privileges. Privilege authorization and authorized privileges pairs indicate process privileges during the execution of the command corresponding to the authorization that the parent process possesses. The authorization and its corresponding privileges are separated by an equal sign (=); individual privileges are separated by a plus sign (+). The attribute is of the **SEC_LIST** type and the value is a null-separated list, so authorization and privileges pairs are separated by a NULL character (\0), as shown in the following illustration: `auth=priv+priv+...\0auth=priv+priv+...\0...\0\0` The number of authorization and privileges pairs is limited to sixteen.	**SEC_LIST**
S_AUTHROLES	A role or list of roles, users having these roles have to be authenticated to allow execution of the command.	**SEC_LIST**
S_INNATEPRIVS	Innate privileges. This is a null-separated list of privileges that are assigned to the process when running the command.	**SEC_LIST**
S_INHERITPRIVS	Inheritable privileges. This is a null-separated list of privileges that are assigned to child processes.	**SEC_LIST**
S_EUID	The effective user ID to be assumed when running the command.	**SEC_INT**
S_EGID	The effective user ID to be assumed when running the command.	**SEC_INT**
S_RUID	The real user ID to be assumed when running the command.	**SEC_INT**

Note: All the above fields corresponds to the **attr_name** attribute.

Item	Description
attr_idx	This attribute is used internally by the **putcmdattrs** subroutine.
attr_type	The type of the attribute that is being updated.
attr _flag	The result of the request to update the target attribute. On successful completion, a value of zero is returned. Otherwise , it returns a value of nonzero.
	A union that contains the value to update the requested attribute with.
attr_domain	This field is ignored by the **putcmdattrs** subroutine.

The following union members that correspond to the definitions of the **attr_char**, **attr_int**, **attr_long** and **attr_llong** macros in the **usersec.h** file respectively.

Item	Description
au_char	A character pointer to the value that is to be written for attributes of the **SEC_CHAR** and **SEC_LIST** types.
au_int	Integer value that is to be written for attributes of the **SEC_INT** type.
au_long	Long value that is to be written for attributes of the **SEC_LONG** type.
au_llong	Long long value that is to be written for attributes of the **SEC_LLONG** type.

Parameters

Item	Description
Command	Specifies the command name for which the attributes are to be updated.
Attributes	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of command attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the *Attributes* parameter.

Security

Files Accessed:

File	Mode
/etc/security/privcmds	rw

Return Values

If the command specified by the *Command* parameter exists in the privileged command database, the **putcmdattrs** subroutine returns zero, even in the case when no attributes in the *Attributes* parameter were successfully updated. On success, the **attr_flag** attribute of each element in the *Attributes* parameter must be examined to determine if it was successfully updated. On failure, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **putcmdattrs** subroutine returns -1, one of the following **errno** values can be set:

Item	Description
EINVAL	The *Command* parameter is **NULL**, **default** or **ALL**.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Attributes* parameter is **NULL** and the *Count* parameter is greater than zero.
EINVAL	The *Attributes* parameter does not point to valid data for the requested attribute.
ENOENT	The command specified in the *Command* parameter does not exist.
EPERM	The operation is not permitted.

If the **putcmdattrs** subroutine fails to update an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The invoker does not have write access to the privileged command database.
EINVAL	The **attr_name** field in the *Attributes* entry is not a recognized command attribute.
EINVAL	The **attr_type** field in the *Attributes* entry contains an invalid type.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute.

putconfattrs Subroutine

Purpose

Accesses system information in the system information database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
#include <userconf.h>

int putconfattrs (Table, Attributes, Count)
char * Table;
dbattr_t * Attributes;
int Count
```

Description

The **putconfattrs** subroutine writes one or more attributes into the system information database. If the database is not already open, the subroutine does an implicit open for reading and writing. Data changed by **putconfattrs** must be explicitly committed by calling the **putconfattr** subroutine with a *Type* parameter specifying the **SEC_COMMIT** value. Until the data is committed, only **get** subroutine calls within the process return the written data.

The *Attributes* array contains information about each attribute that is to be written. The **dbattr_t** data structure contains the following fields:

attr_name
The name of the desired attribute.

attr_idx
Used internally by the **putconfattrs** subroutine.

attr_type
The type of the desired attribute. The list of attribute types is defined in the **usersec.h** header file.

attr_flag
The results of the request to write the desired attribute.

attr_un
A union containing the values to be written. Its union members that follow correspond to the definitions of the **attr_char**, **attr_int**, **attr_long**, and **attr_llong** macros, respectively:

au_char
Attributes of type **SEC_CHAR** and **SEC_LIST** store a pointer to the value to be written.

au_int
Attributes of type **SEC_INT** and **SEC_BOOL** contain the value of the attribute to be written.

au_long
Attributes of type **SEC_LONG** contain the value of the attribute to be written.

au_llong
Attributes of type **SEC_LLONG** contain the value of the attribute to be written.

attr_domain
The authentication domain containing the attribute. The **putconfattrs** subroutine stores the name of the authentication domain that was used to write this attribute if it is not initialized by the caller. The **putconfattrs** subroutine is responsible for managing the memory referenced by this pointer.

Use the **setuserdb** and **enduserdb** subroutines to open and close the system information database. Failure to explicitly open and close the system information database can result in loss of memory and performance.

Parameters

Item	Description
Table	The system information table containing the desired attributes. The list of valid system information tables is defined in the **userconf.h** header file.
Attributes	A pointer to an array of one or more elements of type **dbattr_t**. The list of system attributes is defined in the **usersec.h** header file.
Count	The number of array elements in *Attributes*.

Security

Files accessed:

Item	Description
Mode	**File**

Item	Description
rw	**/etc/security/.ids**
rw	**/etc/security/audit/config**
rw	**/etc/security/audit/events**
rw	**/etc/security/audit/objects**
rw	**/etc/security/login.cfg**
rw	**/etc/security/portlog**
rw	**/etc/security/roles**
rw	**/usr/lib/security/methods.cfg**
rw	**/usr/lib/security/mkuser.sys**

Return Values

The **putconfattrs** subroutine, when successfully completed, returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **putconfattrs** subroutine fails if one or more of the following are true:

Item	Description
EACCES	The system information database could not be accessed for writing.
EINVAL	The *Table* parameter is the NULL pointer.
EINVAL	The *Attributes* parameter does not point to valid data for the requested attribute. Limited testing is possible and all errors might not be detected.
EINVAL	The *Count* parameter is less than or equal to 0.
ENOENT	The specified *Table* does not exist.

If the **putconfattrs** subroutine fails to write an attribute, one or more of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The user does not have access to the attribute specified in the **attr_name** field.
EINVAL	The **attr_type** field in the *Attributes* entry contains an invalid type.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
ENOATTR	The **attr_name** field in the *Attributes* entry specifies an attribute that is not defined for this system table.

putdevattr Subroutine

Purpose

Modifies the device security information in the privileged device database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putdevattr (Device, Attribute, Value, Type)
    char *Device;
    char *Attribute;
    void *Value;
    int Type;
```

Description

The **putdevattr** subroutine writes a specified attribute into the device database. If the database is not open, this subroutine does an implicit open for reading and writing. Data changed by the **putdevattr** and **putdevattrs** subroutines must be explicitly committed by calling the **putdevattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the subroutines within the process return written data.

New entries in the device databases must first be created by invoking the **putdevattr** subroutine with the **SEC_NEW** type.

Changes that are made to the privileged device database do not impact security considerations until the entire database is sent to the Kernel Security Tables through the **setkst** device or until the system is rebooted.

Parameters

Item	Description
Device	The device name. The value should be the full path to the device on the system. This parameter must be specified unless the *Type* parameter is **SEC_COMMIT**.
Attribute	Specifies that attribute is written. The following possible attributes are defined in the **usersec.h** file: **S_READPRIVS** Privileges required to read from the device. Eight privileges can be defined. A process with any of the read privileges is allowed to read from the device. The attribute type is **SEC_LIST**. **S_WRITEPRIVS** Privileges required to write to the device. Eight privileges can be defined. A process with any of the write privileges is allowed to write to the device. The attribute type is **SEC_LIST**.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Item	Description
Type	Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include: **SEC_INT** The format of the attribute is an integer. The user should supply an integer. **SEC_CHAR** The format of the attribute is a null-terminated character string. The user should supply a character pointer. **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters. The user should supply a character pointer. **SEC_COMMIT** Specified that changes to the named device are to be committed to permanent storage. The *Attribute* and *Value* parameters are ignored. If no device is specified, the changes to all modified devices are committed to permanent storage. **SEC_DELETE** If the *Attribute* parameter is specified, the corresponding attribute is deleted from the privileged device database. If no *Attribute* parameter is specified, the entire device definition is deleted from the privileged device database. **SEC_NEW** Creates a new device in the privileged device database when it is specified with the **putdevattr** subroutine.

Security

Files Accessed:

File	Mode
/etc/security/privdevs	rw

Return Values

If successful, the **putdevattr** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **putdevattr** subroutine fails, one of the following **errno** values can be set:

Item	Description
EINVAL	The *Device* parameter is **NULL** and the *Type* parameter is not **SEC_COMMIT**.
EINVAL	The *Device* parameter is **default** or **ALL**.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or is **NULL**.
EINVAL	The *Type* parameter does not contain one of the defined values.
EINVAL	The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
ENOENT	The device specified by the *Device* parameter does not exist.
EPERM	The operation is not permitted.

putdevattrs Subroutine

Purpose

Modifies multiple device attributes in the privileged device database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putdevattrs(Device, Attributes, Count)
    char *Device;
    dbattr_t *Attributes;
    int Count;
```

Description

The **putdevattrs** subroutine modifies one or more attributes from the privileged device database. If the database is not open, this subroutine does an implicit open for reading and writing. The device specified by the *Device* parameter must include the full path to the device and exist in the privileged device database.

The **putdevattrs** subroutine is only used to modify attributes of existing devices in the database. To create or remove device entries, use the **putdevattr** subroutine instead. Data changed by the **putdevattrs** subroutine must be explicitly committed by calling the **putdevattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getdevattr** and **getdevattrs** subroutines within the process return the modified data. Changes made to the privileged device database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** device.

The *Attributes* parameter contains information about each attribute that is to be updated. Each value specified in the *Attributes* parameter must be examined on a successful call to the **putdevattrs** subroutine to determine if the *Attributes* parameter was successfully written. The **dbattr_t data** structure contains the following fields:

Item	Description
attr_name	The name of the device attribute to update.
attr_idx	This attribute is used internally by the **putdevattrs** subroutine.
attr_type	The type of the attribute being updated.
attr _flag	The result of the request to update the desired attribute. On success, a value of zero is returned. Otherwise, a nonzero value is returned.
attr_un	A union containing the value to update the requested attribute with.
attr_domain	This field is ignored by the **putdevattrs** subroutine.

The following valid privileged device attributes for the **putdevattrs** subroutine are defined in the **usersec.h** file:

Name	Description	Type
S_READPRIVS	Privileges required to read from the device. Eight privileges can be defined. A process with any of the read privileges is allowed to read from the device.	**SEC_LIST**
S_WRITEPRIVS	Privileges required to write to the device. Eight privileges can be defined. A process with any of the write privileges is allowed to write to the device.	**SEC_LIST**

The union members that follow correspond to the definitions of the **attr_char, attr_int, attr_long** and **attr_llong** macros in the **usersec.h** file respectively.

Item	Description
au_char	A character pointer to the value to be written for attributes of the **SEC_CHAR** and **SEC_LIST** types.
au_int	Integer value to be written for attributes of the **SEC_INT** type.
au_long	Long value to be written for attributes of the **SEC_LONG** type.
au_llong	Long long value to be written for attributes of the **SEC_LLONG** type.

Parameters

Item	Description
Device	Specifies the device name for which the attributes are to be updated.
Attributes	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of device attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the *Attributes* parameter.

Security

Files Accessed:

File	Mode
/etc/security/privdevs	rw

Return Values

If the device specified by the *Device* parameter exists in the privileged device database, the **putdevattrs** subroutine returns zero, even in the case when no attributes in the *Attributes* parameter were successfully updated. On success, the **attr_flag** attribute of each element in the *Attributes* parameter must be examined to determine if it was successfully updated. On failure, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **putdevattrs** subroutine returns -1, one of the following **errno** values can be set:

Item	Description
EINVAL	The *Device* parameter is **NULL**, **default** or **ALL**.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Attributes* parameter is **NULL** and the *Count* parameter is greater than zero.
EINVAL	The *Attributes* parameter does not point to valid data for the requested attribute.
ENOENT	The device specified in the *Device* parameter does not exist.
EPERM	The operation is not permitted.

If the **putdevattrs** subroutine fails to update an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding to the value specified by the Attributes entry:

Item	Description
EACCES	The invoker does not have write access to the privileged device database.
EINVAL	The **attr_name** field in the *Attributes* entry is not a recognized privileged device attribute.
EINVAL	The **attr_type** field in the *Attributes* entry contains a type that is not valid.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute.

putdomattr Subroutine

Purpose

Modifies the domains that are defined in the domain database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int putdomattr ( Dom, Attributes, Value, Type)
char * Dom;
char * Attribute;void * Value;
int Type;
```

Description

The **putdomattr** subroutine modifies the domain database.

New domains can be added to the domain database by calling the **putdomattr** subroutine with the SEC_NEW type and specifying the new domain name. Deletion of a domain or domain attribute is done using the SEC_DELETE type for the **putdomattr** subroutine. Data changed by the **putdomattr** subroutine must be explicitly committed by calling the **putdomattr** subroutine with a Type parameter specifying the SEC_COMMIT type. Until all the data is committed, only the **getdomattr** and **getdomattrs** subroutines within the process return the modified data. Changes that are made to the domain database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command or until the system is rebooted.

Parameters

Item	Description
Dom	The domain name. This parameter must be specified unless the Type parameter is SEC_COMMIT.
Attribute	Specifies the attribute to be written. The following possible attributes are defined in the **usersec.h** file: **S_DFLTMSG** Specifies a default domain description to use if message catalogs are not in use. The attribute type is SEC_CHAR. **S_ID** Specifies a unique integer that is used to identify the domain. The attribute type is SEC_INT. **Note:** Do not modify this value after it is set initially when the domain is created. Modifying the value might compromise the security of the system. **S_MSGCAT** Specifies the message catalog file name that contains the description of the domain. The attribute type is SEC_CHAR. **S_MSGSET** Specifies the message set that contains the message for the description of the domain in the file specified by the S_MSGCAT attribute. The attribute type is SEC_INT. **S_MSGNUMBER** Specifies the message number for the description of the domain in the file that is specified by the S_MSGCAT attribute and the message set that is specified by the S_MSGSET attribute. The attribute type is SEC_INT.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer according to the values of the *Attribute* and *Type* parameters. See the *Type* parameter for more details. Specifies the type of attribute. The following valid types are defined in the **usersec.h** file: **SEC_INT** The format of the attribute is an integer. The user should supply an integer value. **SEC_CHAR** The format of the attribute is a null-terminated character string. The user should supply a character pointer.

Item	Description
Type	**SEC_LIST** The format of the attribute is a series of concatenated strings, each of which is null-terminated. The last string in the series is terminated by two successive null characters. The user should supply a character pointer. **SEC_COMMIT** Specifies that the changes to the named domain are to be committed to permanent storage. The values of the Attribute and Value parameters are ignored. If no domain is specified, the changes to all modified domains are committed to permanent storage. **SEC_DELETE** If the Attribute parameter is specified, the corresponding attribute is deleted from the domain database. If no Attribute parameter is specified, the entire domain definition is deleted from the domain database. **SEC_NEW** Creates a new domain in the domain database.

Security

Files Accessed:

Item	Description
File	**Mode**
`/etc/security/domains`	`rw`

Return Values

If successful, the **putdomattr** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

Item	Description
EINVAL	The Dom parameter is NULL and the Type parameter is not SEC_COMMIT. The Dom parameter is default or ALL The Attribute parameter is NULL and the Type parameter is not SEC_NEW, SEC_DELETE or SEC_COMMIT. The Attribute parameter does not contain one of the defined attributes. The Type parameter does not contain one of the defined values. The Value parameter does not point to a valid buffer or to valid data for this type of attribute.
ENOENT	The domain specified in the *Dom* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.

putdomattrs Subroutine

Purpose

Modifies multiple domain attributes in the domain-assigned object database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int putdomattrs ( Dom, Attributes, Count)
char * Dom;
dbattr_t * Attributes;
int Count;
```

Description

The **putdomattrs** subroutine modifies one or more attributes from the domain-assigned object database. The subroutine can be called only with an domain that already exists in the domain-assigned object database.

To create or remove domains, use the **putdomattr** subroutine instead. Data changed by the **putdomattrs** subroutine must be explicitly committed by calling the **putdomattr** subroutine with a Type parameter specifying SEC_COMMIT. Until the data is committed, only the **getdomattr** and **getdomattrs** subroutines within the process return the modified data. Changes that are made to the domain database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command. The *Attributes* array contains information about each attribute that is to be updated. Each value specified in the Attributes array must be examined on a successful call to the **putdomattrs** subroutine to determine whether the value of the *Attributes* array was successfully written. The **dbattr_t** data structure contains the following fields:

Item	Description
attr_name	The name of the domain attribute to update.
attr_idx	This attribute is used internally by the **putdomattrs** subroutine.
attr_type	The type of the attribute that is being updated.
attr _flag	The result of the request to update the target attribute. On successful completion, a value of zero is returned. Otherwise, a value of nonzero value is returned. A union that contains the value to update the requested attribute with.
attr_domain	This field is ignored by the **putdomattrs** subroutine.

The following valid domain attributes for the **putdomattrs** subroutine are defined in the **usersec.h** file:

Name	Description	Type
S_DFLTMSG	The default domain description that is used when catalogs are not in use. A unique integer that is used to identify the domain.	SEC_CHAR
S_ID	**Note:** After the value is set initially, it must not be modified because it might be in use on the system.	SEC_INT
S_MSGCAT	The message catalog name that contains the domain description.	SEC_CHAR

Name	Description	Type
S_MSGSET	The message catalog's set number for the domain description.	SEC_INT
S_MSGNUMBER	The message number for the domain description.	SEC_INT

The following union members correspond to the definitions of the ATTR_CHAR, ATTR_INT, ATTR_LONG and the ATTR_LLONG macros in the **usersec.h** file respectively.

Item	Description
au_char	A character pointer to the value that is to be written for attributes of SEC_CHAR and SEC_LIST types.
au_int	Integer value that is to be written for attributes of the SEC_INT type.
au_long	Long value that is to be written for attributes of the SEC_LONG type.
au_llong	Long long value that is to be written for attributes of the SEC_LLONG type.

Parameters

Item	Description
Dom	Specifies the domain name for which the attributes are to be updated.
Attribute	A pointer to an array of zero or more attributes of the **dbattr_t** type. The list of domain attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the *Attribute* parameter.

Security

Files Accessed:

File	Mode
/etc/security/domains	**rw**

Return Values

If the domain specified by the *Dom* parameter exists in the domain database, the **putdomattrs** subroutine returns zero, even in the case when no attributes in the **Attributes** array are successfully updated. On successful completion, the **attr_flag** attribute of each value that is specified in the **Attributes** array must be examined to determine whether it was successfully updated. If the specified domain does not exist, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

Item	Description
EINVAL	The *Dom* parameter is NULL or default. The *Count* parameter is less than zero. The Attributes array is NULL and the Count parameter is greater than zero. The Attributes array does not point to valid data for the requested attribute.
ENOENT	The domain specified in the *Dom* parameter does not exist.

Item	Description
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

If the **putdomattrs** subroutine fails to update an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The invoker does not have write access to the domain database.
EINVAL	The **attr_name** field in the Attributes entry is not a recognized domain attribute. The **attr_type** field in the Attributes entry contains a type that is not valid. The **attr_un** field in the Attributes entry does not point to a valid buffer or to valid data for this type of attribute.

putenv Subroutine

Purpose

Sets an environment variable.

Library

Standard C Library (**libc.a**)

Syntax

```
int putenv ( String)
char *String;
```

Description

Attention: Unpredictable results can occur if a subroutine passes the **putenv** subroutine a pointer to an automatic variable and then returns while the variable is still part of the environment.

The **putenv** subroutine sets the value of an environment variable by altering an existing variable or by creating a new one. The *String* parameter points to a string of the form *Name*=*Value*, where *Name* is the environment variable and *Value* is the new value for it.

The memory space pointed to by the *String* parameter becomes part of the environment, so that altering the string effectively changes part of the environment. The space is no longer used after the value of the environment variable is changed by calling the **putenv** subroutine again. Also, after the **putenv** subroutine is called, environment variables are not necessarily in alphabetical order.

The **putenv** subroutine manipulates the **environ** external variable and can be used in conjunction with the **getenv** subroutine. However, the *EnvironmentPointer* parameter, the third parameter to the main subroutine, is not changed.

The **putenv** subroutine uses the **malloc** subroutine to enlarge the environment.

Parameters

Item	Description
String	A pointer to the *Name*=*Value* string.

Return Values

Upon successful completion, a value of 0 is returned. If the **malloc** subroutine is unable to obtain sufficient space to expand the environment, then the **putenv** subroutine returns a nonzero value.

putgrent Subroutine

Purpose

Updates group descriptions.

Library

Standard C Library (**libc.a)**

Syntax

```
int putgrent (grp, fp)
struct group *grp;
FILE *fp;
```

Description

The **putgrent** subroutine updates group descriptions. The *grp* parameter is a pointer to a group structure, as created by the **getgrent**, **getgrgid**, and **getgrnam** subroutines.

The **putgrent** subroutine writes a line on the stream specified by the *fp* parameter. The stream matches the format of **/etc/group**.

The **gr_passwd** field of the line written is always set to ! (exclamation point).

Parameters

Item	Description
grp	Pointer to a group structure.
fp	Specifies the stream to be written to.

Return Values

The **putgrent** subroutine returns a value of 0 upon successful completion. If **putgrent** fails, a nonzero value is returned.

Files

/etc/group

/etc/security/group

putgroupattrs Subroutine

Purpose

Stores multiple group attributes in the group database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putgroupattrs (Group, Attributes, Count)
char * Group;
dbattr_t * Attributes;
int Count
```

Description

The **putgroupattrs** subroutine writes multiple group attributes into the group database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by **putgroupattrs** must be explicitly committed by calling the **putgroupattr** subroutine with a *Type* parameter specifying the **SEC_COMMIT** value. Until the data is committed, only **get** subroutine calls within the process return the written data.

The *Attributes* array contains information about each attribute that is to be written. Each element in the *Attributes* array must be examined upon a successful call to **putgroupattrs** to determine if the *Attributes* array entry was successfully put. The **dbattr_t** data structure contains the following fields:

attr_name
The name of the desired attribute.

attr_idx
Used internally by the **putgroupattrs** subroutine.

attr_type
The type of the desired attribute. The list of attribute types is defined in the **usersec.h** header file.

attr_flag
The results of the request to write the desired attribute.

attr_un
A union containing the values to be written. Its union members that follow correspond to the definitions of the **attr_char**, **attr_int**, **attr_long**, and **attr_llong** macros, respectively:

au_char
Attributes of type **SEC_CHAR** and **SEC_LIST** store a pointer to the value to be written.

au_int
Attributes of type **SEC_INT** and **SEC_BOOL** contain the value of the attribute to be written.

au_long
Attributes of type **SEC_LONG** contain the value of the attribute to be written.

au_llong
Attributes of type **SEC_LLONG** contain the value of the attribute to be written.

attr_domain
The authentication domain containing the attribute. The **putgroupattrs** subroutine stores the name of the authentication domain that was used to write this attribute if it is not initialized by the caller. The **putgroupattrs** subroutine is responsible for managing the memory referenced by this pointer. If **attr_domain** is specified for an attribute, the put request is sent only to that domain. If **attr_domain**

is not specified (that is, set to NULL), **putgroupattrs** attempts to put the attributes to the first domain associated with the user. All put requests for the attributes with a NULL **attr_domain** are sent to the same domain. In other words, values cannot be put into different domains where **attr_domain** is unspecified; **attr_domain** is set to the name of the domain where the value is put and returned to the invoker. When **attr_domain** is not specified, the list of searchable domains can be restricted to a particular domain by using the **setauthdb** function call.

Use the **setuserdb** and **enduserdb** subroutines to open and close the group database. Failure to explicitly open and close the group database can result in loss of memory and performance.

Parameters

Item	Description
Group	Specifies the name of the group for which the attributes are to be written.
Attributes	A pointer to an array of one or more elements of type **dbattr_t**. The list of group attributes is defined in the **usersec.h** header file.
Count	The number of array elements in *Attributes*.

Security

Files accessed:

Item	Description
Mode	**File**
rw	**/etc/group**
rw	**/etc/security/group**
rw	**/etc/security/smitacl.group**

Return Values

The **putgroupattrs** subroutine returns a value of 0 if the *Group* exists, even in the case when no attributes in the *Attributes* array were successfully updated. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes

The **putgroupattrs** subroutine fails if one or more of the following are true:

Item	Description
EACCES	The system information database could not be accessed for writing.
EINVAL	The *Group* parameter is the NULL pointer.
EINVAL	The *Attributes* parameter does not point to valid data for the requested attribute. Limited testing is possible and all errors might not be detected.
EINVAL	The *Count* parameter is less than or equal to 0.
ENOENT	The specified *Group* does not exist.

If the **putgroupattrs** subroutine fails to write an attribute, one or more of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The user does not have access to the attribute specified in the **attr_name** field.

Item	Description
EINVAL	The **attr_type** field in the *Attributes* entry contains an invalid type.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
ENOATTR	The **attr_name** field in the *Attributes* entry specifies an attribute that is not defined for this group.

Examples

The following sample test program displays the output to a call to **putgroupattrs**. In this example, the system has a user named `foo` and a group named `bar`.

```
#include <stdio.h>
#include <strings.h>
#include <string.h>
#include <usersec.h>


char * CommaToNSL(char *);

#define NATTR         2       /* Number of attributes to be put. */
#define GROUPNAME    "bar"    /* Group name. */
#define DOMAIN        "files"  /* Domain where attributes are going to put. */

main(int argc, char *argv[]) {
    int      rc;
    int      i;
    dbattr_t attributes[NATTR];

        /* Open the group database */
        setuserdb(S_WRITE);

    /* Valid put */

    attributes[0].attr_name = S_ADMIN;
        attributes[0].attr_type = SEC_BOOL;
    attributes[0].attr_domain = DOMAIN;
    attributes[0].attr_char = strdup("false");

    /* Valid put */

    attributes[1].attr_name = S_USERS;
    attributes[1].attr_type = SEC_LIST;
    attributes[1].attr_domain = DOMAIN;
    attributes[1].attr_char = CommaToNSL("foo");


    rc = putgroupattrs(GROUPNAME, attributes, NATTR);

    if (rc) {
        printf("putgroupattrs failed \n");
        goto clean_exit;
    }

    for (i = 0; i < NATTR; i++) {
        if (attributes[i].attr_flag)
            printf("Put failed for attribute %s. errno = %d \n",
                attributes[i].attr_name, attributes[i].attr_flag);
        else
            printf("Put succeded for attribute %s \n",
                attributes[i].attr_name);
    }

clean_exit:
        enduserdb();

    if (attributes[0].attr_char)
                free(attributes[0].attr_char);

        if (attributes[1].attr_char)
                free(attributes[1].attr_char);
```

```
    exit(rc);
}

/*
 * Returns a new NSL created from a comma separated list.
 * The comma separated list is unmodified.
 *
 */
char *
CommaToNSL(char *CommaList)
{
        char    *NSL = (char *) NULL;
        char    *s;

        if (!CommaList)
            return(NSL);

        if (!(NSL = (char *) malloc(strlen(CommaList) + 2)))
            return(NSL);

        strcpy(NSL, CommaList);

        for (s = NSL; *s; s++)
                if (*s == ',')
                        *s = '\0';

        *(++s) = '\0';
}
```

The following output for the call is expected:

```
Put succeeded for attribute admin
Put succeeded for attribute users
```

putobjattr Subroutine

Purpose

Modifies the object that are defined in the domain-assigned object database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int putobjattr ( Obj, Attribute, Value, Type )
char * Obj;
char *Attribute;
void * Value;
int Type;
```

Description

The **putobjattr** subroutine modifies the domain-assigned object database. New object can be added to the domain-assigned object database by calling the **putobjattr** subroutine with the SEC_NEW type and specifying the new object name. Deletion of an object or object attribute is done using the SEC_DELETE type for the **putobjattr** subroutine.

Data changed by the **putobjattr** subroutine must be explicitly committed by calling the **putobjattr** subroutine with a *Type* parameter specifying the SEC_COMMIT type. Until all the data is committed, only the **getobjattr** and **getobjattrs** subroutines within the process return the modified data. Changes that are made to the domain database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command or until the system is rebooted.

Parameters

Item	Description
Obj	The object name. This parameter must be specified unless the Type parameter is SEC_COMMIT.
Attribute	Specifies the attribute to be written. The following possible attributes are defined in the **usersec.h** file: • S_DOMAINS The list of domains to which the object belongs. The attribute type is SEC_LIST. • S_CONFSETS The list of domains that are excluded from accessing the object. The attribute type is SEC_LIST. • S_OBJTYPE The type of the object. Valid values are: – S_NETINT For network interfaces – S_FILE For file based objects. The object name should be the absolute path – S_DEVICE For Devices. The absolute path should be specified. – S_NETPORT For port and port ranges The attribute type is SEC_CHAR **S_SECFLAGS** The security flags for the object. The valid values are FSF_DOM_ALL and FSF_DOM_ANY. The attribute type is SEC_INT
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer according to the values of the *Attribute* and *Type* parameters. See the *Type* parameter for more details.

Item	Description
Type	Specifies the type of the attribute. The following valid types are defined in the **usersec.h** file: • SEC_INT The format of the attribute is an integer. You should supply an integer value. • SEC_CHAR The format of the attribute is a null-terminated character string. You should supply a character pointer. • SEC_LIST The format of the attribute is a series of concatenated strings, each of which is null-terminated. The last string in the series is terminated by two successive null characters. You should supply a character pointer. • SEC_COMMIT Specifies that the changes to the named objects that are to be committed to the permanent storage. The values of the *Attribute* and *Value* parameters are ignored. If no object is specified, the changes to all modified objects are committed to the permanent storage. • SEC_DELETE If the *Attribute* parameter is specified, the corresponding attribute is deleted from the object database. If no *Attribute* parameter is specified, the entire object definition is deleted from the domain-assigned object database. • SEC_NEW Creates a new object in the domain-assigned object database.

Security

Files Accessed:

Item	Description
File	**Mode**
`/etc/security/domobjs`	`rw`

Return Values

If successful, the **putobjattr** subroutine returns zero. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **putobjattr** subroutine fails, one of the following **errno** values is set:

Item	Description
EINVAL	The *Obj* parameter is NULL and the *Type* parameter is not SEC_COMMIT. The *Obj* parameter is default or ALL The *Attribute* parameter is NULL and the *Type* parameter is not SEC_NEW, SEC_DELETE or SEC_COMMIT. The *Attribute* parameter does not contain one of the defined attributes. The *Type* parameter does not contain one of the defined values. The *Value* parameter does not point to a valid buffer or to valid data for this type of attribute.
ENOENT	The object specified by the *Obj* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.

putobjattrs Subroutine

Purpose

Modifies the multiple object security attributes in the domain-assigned object database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
int putobjattrs ( Obj, Attributes,Count )
char * Dom;
dbattr_t *Attributes;
intCount;
```

Description

The **putobjattrs** subroutine modifies one or more attributes from the domain-assigned object database. The subroutine can be called only with an object that already exists in the domain-assigned object database.

To create or remove an object, use the **putobjattr** subroutine instead. Data changed by the **putobjattrs** subroutine must be explicitly committed by calling the **putobjattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until the data is committed, only the **getobjattr** and **getobjattrs** subroutines within the process return the modified data.

Changes that are made to the domain object database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command.

The **Attributes** array contains information about each attribute that is to be updated. Each value specified in the **Attributes** array must be examined on a successful call to the **putobjattrs** subroutine to determine whether the value of the **Attributes** array was successfully written. The **dbattr_t** data structure contains the following fields:

Item	Description
attr_name	Specifies the name.

Item	Description
attr_idx	This attribute is used internally by the **putobjattrs** subroutine.
attr_type	The type of the attribute that is being updated.
attr _flag	The result of the request to update the target attribute. On successful completion, a value of zero is returned. Otherwise, a nonzero value is returned. A union that contains the value to update the requested attribute with.

The following table lists the different vales for *attr_name* attribute:

Name	Description	Type
S_DOMAINS	The list of domains to which the object belongs.	SEC_LIST
S_CONFSETS	The list of domains that are excluded from accessing the object.	SEC_LIST
S_OBJTYPE	The type of the object. Valid values are: • S_NETINT For network interfaces • S_FILE For file based objects. The object name should be the absolute path. • S_DEVICE For Devices. The absolute path should be specified. • S_NETPORT For port and port ranges	SEC_CHAR
S_SECFLAGS	The security flags for the object. The valid values are FSF_DOM_ALL and FSF_DOM_ANY.	SEC_INT

The following union members correspond to the definitions of the **attr_char**, **attr_int**, **attr_long** and the **attr_long** macros in the **usersec.h** file respectively.

Item	Description
au_char	A character pointer to the value that is to be written for attributes of SEC_CHAR and SEC_LIST types.
au_int	Integer value that is to be written for attributes of the SEC_INT type.
au_long	Long value that is to be written for attributes of the SEC_LONG type.

Item	Description
au_llong	Long long value that is to be written for attributes of the SEC_LLONG type.

Parameters

Item	Description
Obj	Specifies the domain-assigned object name for which the attributes are to be updated.
Attributes	A pointer to an array of zero or more attributes of the **dbattr_t** type. The list of domain-assigned object attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the *Attributes* parameter.

Security

Files Accessed:

Item	Description
File	**Mode**
`/etc/security/domobjs`	`rw`

Return Values

If the object specified by the *Obj* parameter exists in the domain-assigned object database, the **putobjattrs** subroutine returns zero, even in the case when no attributes in the **Attributes** array are successfully updated. On successful completion, the **attr_flag** attribute that is specified in the **Attributes** array must be examined to determine whether it was successfully updated. If the specified object does not exist, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **putobjattrs** returns -1, one of the following **errno** values is set:

Item	Description
EINVAL	The *Obj* parameter is NULL or default. The *Count* parameter is less than zero. The **Attributes** array is NULL and the *Count* parameter is greater than zero. The **Attributes** array does not point to valid data for the requested attribute.
ENOENT	The object specified by the *Obj* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

If the **putobjattrs** subroutine fails to update an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding **Attributes** element:

Item	Description
EINVAL	The **attr_name** field in the **Attributes** entry is not a recognized object attribute. The **attr_type** field in the **Attributes** entry contains a type that is not valid. The **attr_un** field in the **Attributes** entry does not point to a valid buffer or to valid data for this type of attribute.
EACCES	The caller does not have write access to the domain database.

putp, tputs Subroutine

Purpose

Outputs commands to the terminal.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int putp(const char *str);

int tputs(const char *str,
int affcnt,
int (*putfunc)(int));
```

Description

These subroutines output commands contained in the terminfo database to the terminal.

The **putp** subroutine is equivalent to **tputs**(str, 1, putchar). The output of the **putp** subroutine always goes to stdout, not to the fildes specified in the **setupterm** subroutine.

The **tputs** subroutine outputs *str* to the terminal. The *str* argument must be a terminfo string variable or the return value from the **tgetstr**, **tgoto**, **tigestr**, or **tparm** subroutines. The *affcnt* argument is the number of lines affected, or *1* if not applicable. If the terminfo database indicates that the terminal in use requires padding after any command in the generated string, the **tputs** subroutine inserts pad characters into the string that is sent to the terminal, at positions indicated by the terminfo database. The **tputs** subroutine outputs each character of the generated string by calling the user-supplied **putfunc** subroutine (see below).

The user-supplied **putfunc** subroutine (specified as an argument to the **tputs** subroutine is either **putchar** or some other subroutine with the same prototype. The **tputs** subroutine ignores the return value of the **putfunc** subroutine.

Parameters

Item	Description
**str*	
*affcn*t	
**putfunc*	

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

For the **putp** subroutine:

To call the **tputs(**`my_string`**, 1, putchar)** subroutine, enter:

```
char *my_string;
putp(my_string);
```

For the **tputs** subroutine:

1. To output the clear screen sequence using the user-defined **putchar**-like subroutine **my_putchar**, enter:

   ```
   int_my_putchar();
   tputs(clear_screen, 1 ,my_putchar);
   ```

2. To output the escape sequence used to move the cursor to the coordinates x=40, y=18 through the user-defined **putchar**-like subroutine `my_putchar`, enter:

   ```
   int_my_putchar();
   tputs(tparm(cursor_address, 18, 40), 1, my_putchar);
   ```

putpfileattr Subroutine

Purpose

Accesses the privileged file security information in the privileged file database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putpfileattr (File, Attribute, Value, Type)
    char *File;
    char *Attribute;
    void *Value;
    int Type;
```

Description

The **putpfileattr** subroutine writes a specified attribute into the privileged file database. If the database is not open, this subroutine opens the database implicitly for reading and writing. Data changed by the **putpfileattr** and **putpfileattrs** subroutines must be explicitly committed by calling the **putpfileattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only these subroutines within the process return written data.

New entries in the privileged file databases must first be created by invoking the **putpfileattr** subroutine with the **SEC_NEW** type.

Parameters

Item	Description
File	The file name. The value should be the full path to the file on the system. This parameter must be specified unless the *Type* parameter is **SEC_COMMIT**.
Attribute	Specifies which attribute is read. The following possible attributes are defined in the **usersec.h** file: **S_READAUTHS** Authorizations required to read the file using the **pvi** command. A total of eight authorizations can be defined. The attribute type is **SEC_LIST**. **S_WRITEAUTHS** Authorizations required to write to the file using the **pvi** command. A total of eight authorizations can be defined. The attribute type is **SEC_LIST**.
Value	Specifies a buffer, a pointer to a buffer, or a pointer to a pointer depending on the *Attribute* and *Type* parameters. See the *Type* parameter for more details.
Type	Specifies the type of attribute expected. Valid types are defined in the **usersec.h** file and include: **SEC_LIST** The format of the attribute is a series of concatenated strings, each null-terminated. The last string in the series is terminated by two successive null characters. For the **putpfileattr** subroutine, the user should supply a character pointer. **SEC_COMMIT** For the **putpfileattr** subroutine, this value specified by itself indicates that changes to the security attributes of the named file are to be committed to the permanent storage. The *Attribute* and *Value* parameters are ignored. If no file is specified, the changes to all modified files are committed to the permanent storage. **SEC_DELETE** If the *Attribute* parameter is specified, then the corresponding attribute is deleted from the privileged file database. If no *Attribute* parameter is specified, then the entire file definition is deleted from the privileged file database. **SEC_NEW** Creates a new file in the privileged file database when it is specified with the **putpfileattr** subroutine.

Security

Files Accessed:

File	Mode
/etc/security/privfiles	rw

Return Values

If successful, the **putpfileattr** subroutine returns 0. Otherwise, a value of -1 is returned and the **errno** global value is set to indicate the error.

Error Codes

If the **putpfileattr** subroutine fails, one of the following **errno** values can be set:

Item	Description
EINVAL	The *File* parameter is **NULL** and the *Type* parameter is **SEC_NEW** or **SEC_DELETE**.
EINVAL	The *File* parameter is **default** or **ALL**.
EINVAL	The *Attribute* parameter does not contain one of the defined attributes or is **NULL**.
EINVAL	The *Type* parameter does not contain one of the defined values.
EINVAL	The *Value* parameter does not point to a valid buffer or to the valid data for this type of attribute.
ENOENT	The file specified by the *File* parameter does not exist.
EPERM	Operation is not permitted.

putpfileattrs Subroutine

Purpose

Updates multiple file attributes in the privileged files database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putpfileattrs(File, Attributes, Count)
    char *File;
    dbattr_t *Attributes;
    int Count;
```

Description

The **putpfileattrs** subroutine modifies one or more attributes from the privileged files database (**/etc/security/privfiles**). If the database is not open, this subroutine opens the database implicitly for reading and writing. The file specified by the *File* parameter must include the full path to the file and exist in the privileged file database.

The **putpfileattrs** subroutine is only used to modify attributes of existing files in the database. To create or remove file entries, use the **putpfileattr** subroutine instead. Data changed by the **putpfileattrs** subroutine must be explicitly committed by calling the **putpfileattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getpfileattr** and **getpfileattrs** subroutines within the process return the modified data.

The Attributes array contains information about each attribute that is to be updated. Each element in the Attributes array must be examined on a successful call to the **putpfileattrs** subroutine to determine if the Attributes array was successfully written. The **dbattr_t data** structure contains the following fields:

Item	Description
attr_name	The name of the file attribute to update.
attr_idx	This attribute is used internally by the **putpfileattrs** subroutine.
attr_type	The type of the attribute being updated.
attr _flag	The result of the request to update the desired attribute. On success, a value of zero is returned. Otherwise, a nonzero value is returned.

Item	Description
attr_un	A union containing the value to update the requested attribute with.

Valid privileged file attributes for the **putpfileattrs** subroutine defined in the **usersec.h** file are:

Name	Description	Type
S_PRIVFILES	Retrieves all the files in the privileged file database. It is valid only when the *File* parameter is **ALL**.	**SEC_LIST**
S_READAUTHS	Read authorization. It is a null separated list of authorization names. A total of eight authorizations can be specified. A user with any one of the authorizations is allowed to read the file using the privileged editor **/usr/bin/pvi**.	**SEC_LIST**
S_WRITEAUTHS	Write authorization. It is a null separated list of authorization names. A total of eight authorizations can be specified. A user with any one of the authorizations is allowed to write the file using the privileged editor **/usr/bin/pvi**.	**SEC_LIST**

The union members that follow correspond to the definitions of the **attr_char, attr_int, attr_long** and **attr_llong** macros in the **usersec.h** file respectively.

Item	Description
au_char	A character pointer to the value to be written for attributes of the **SEC_CHAR** and **SEC_LIST** types. If the pointer is to the allocated memory, the caller is responsible for freeing the memory.
au_int	Integer value to be written for attributes of the **SEC_INT** type.
au_long	Long value to be written for attributes of the **SEC_LONG** type.
au_llong	Long long value to be written for attributes of the **SEC_LLONG** type.

Parameters

Item	Description
File	Specifies the file name for which the attributes are to be updated.
Attributes	A pointer to an array of none or more than one element of the **dbattr_t** type. The list of file attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the Attributes array.

Security

Files Accessed:

File	Mode
/etc/security/privfiles	rw

Return Values

If the file specified by the *File* parameter exists in the privileged file database, the **putpfileattrs** subroutine returns a value of zero, even when no attributes in the Attributes array were successfully updated. On success, the **attr_flag** attribute of each element in the Attributes array must be examined to determine if it was successfully updated. If the specified file does not exist in the database, a value of -1 is returned and the **errno** value is set to indicate the error.

Error Codes

If the **putpfileattrs** subroutine returns -1, one of the following **errno** values can be set:

Item	Description
EINVAL	The *File* parameter is **NULL**, **default** or **ALL**.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Attributes* parameter is **NULL** and the *Count* parameter is greater than zero.
EINVAL	The *Attributes* parameter does not point to valid data for the requested attribute.
ENOENT	The file specified in the *File* parameter does not exist.
EPERM	The operation is not permitted.

If the **putpfileattrs** subroutine fails to update an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding Attributes element:

Item	Description
EACCES	The invoker does not have write access to the privileged file database.
EINVAL	The **attr_name** field in the Attributes entry is not a recognized privileged file attribute.
EINVAL	The **attr_type** field in the Attributes entry contains an invalid type.
EINVAL	The **attr_un** field in the Attributes entry does not point to a valid buffer or to valid data for this type of attribute.

putroleattrs Subroutine

Purpose

Modifies multiple role attributes in the role database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>

int putroleattrs(Role, Attributes, Count)
    char *Role;
    dbattr_t *Attributes;
    int Count;
```

Description

The **putroleattrs** subroutine modifies one or more attributes from the role database. The role specified by the *Role* parameter must already exist in the role database.

The **putroleattrs** subroutine is used to modify attributes of existing roles only. To create or remove user-defined roles, use the **putroleattr** subroutine instead. Data changed by the **putroleattrs** subroutine must be explicitly committed by calling the **putroleattr** subroutine with a *Type* parameter specifying **SEC_COMMIT**. Until all the data is committed, only the **getroleattr** and **getroleattrs** subroutines within the process return the modified data. Changes made to the role database do not impact security considerations until the entire database is sent to the Kernel Security Tables using the **setkst** command.

The *Attributes* array contains information about each attribute that is to be updated. Each element in the *Attributes* array must be examined on a successful call to the **putroleattrs** subroutine to determine if the *Attributes* array was successfully written. The **dbattr_t** data structure contains the following fields:

Item	Description
attr_name	The name of the role attribute to update.
attr_idx	This attribute is used internally by the **putroleattrs** subroutine.
attr_type	The type of the attribute being updated.
attr _flag	The result of the request to update the desired attribute. Zero is returned on success; a nonzero value is returned otherwise.
attr_un	A union containing the value to update the requested query with.
attr_domain	This field is ignored by the **putroleattrs** subroutine.

Valid role attributes for the **putroleattrs** subroutine defined in the **usersec.h** file are:

Name	Description	Type
S_AUTHORIZATIONS	A list of authorizations assigned to the role.	**SEC_LIST**
S_AUTH_MODE	The authentication to perform when assuming the role through the **swrole** command. Possible values are: **NONE** No authentication is required. **INVOKER** This is the default value. Invokers of the **swrole** command must enter their passwords to assume the role.	**SEC_CHAR**
S_DFLTMSG	The default role description used when catalogs are not in use.	**SEC_CHAR**
S_GROUPS	The groups that a user is suggested to be a member of. It is for informational purposes only.	**SEC_LIST**
S_HOSTSENABLEDROLE	The list of hosts from where the role can be downloaded to the Kernel Role Table.	**SEC_LIST**

Name	Description	Type
S_HOSTSDISABLEDROLE	The list of hosts from where the role cannot be downloaded to the Kernel Role Table.	**SEC_LIST**
S_ID	The role identifier.	**SEC_INT**
S_MSGCAT	The message catalog name containing the role description.	**SEC_CHAR**
S_MSGSET	The message catalog set number for the role description.	**SEC_INT**
S_MSGNUMBER	The message number for the role description.	**SEC_INT**
S_ROLELIST	The list of roles whose authorizations are included in this role.	**SEC_LIST**
S_SCREENS	The SMIT screens that the role can access.	**SEC_LIST**
S_VISIBILITY	An integer that determines whether the role is active or not. Possible values are: **-1** The role is disabled. **0** The role is active but not visible from a GUI. **1** The role is active and visible. This is the default value.	**SEC_INT**

The union members that follow correspond to the definitions of the **attr_char, attr_int, attr_long** and **attr_llong** macros in the **usersec.h** file respectively

Item	Description
au_char	A character pointer to the value to be written for attributes of the **SEC_CHAR** and **SEC_LIST** types.
au_int	Integer value to be written for attributes of the **SEC_INT** type.
au_long	Long value to be written for attributes of the **SEC_LONG** type.
au_llong	Long long value to be written for attributes of the **SEC_LLONG** type.

Parameters

Item	Description
Role	Specifies the role name for which the attributes are to be updated.
Attributes	A pointer to an array of zero or more elements of the **dbattr_t** type. The list of role attributes is defined in the **usersec.h** header file.
Count	The number of array elements in the Attributes array.

Security

Files Accessed:

File	Mode
/etc/security/roles	rw

Return Values

If the role specified by the *Role* parameter exists in the role database, the **putroleattrs** subroutine returns zero, even in the case when no attributes in the *Attributes* array were successfully updated. On success, the **attr_flag** attribute of each element in the *Attributes* array must be examined to determine whether it was successfully updated. If the specified role does not exist, a value of -1 is returned, and the **errno** value is set to indicate the error.

Error Codes

If the **putroleattrs** returns -1, one of the following **errno** values can be set:

Item	Description
EINVAL	The *Role* parameter is **NULL** or **ALL**.
EINVAL	The *Count* parameter is less than zero.
EINVAL	The *Attributes* parameter is **NULL** and the *Count* parameter is greater than zero.
EINVAL	The *Attributes* parameter does not point to valid data for the requested attribute.
ENOENT	The role specified by the *Role* parameter does not exist.
ENOMEM	Memory cannot be allocated.
EPERM	The operation is not permitted.
EACCES	Access permission is denied for the data request.

If the **putroleattrs** subroutine fails to update an attribute, one of the following errors is returned in the **attr_flag** field of the corresponding Attributes element:

Item	Description
EACCES	The invoker does not have write access to the role database.
EINVAL	The **attr_name** field in the *Attributes* entry is not a recognized role attribute.
EINVAL	The **attr_type** field in the *Attributes* entry contains a type that is not valid.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute.

puts or fputs Subroutine

Purpose

Writes a string to a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include  <stdio.h>

int   puts   ( String)
const char   *String;
```

```
int   fputs  (String,  Stream)
const char   *String;
FILE         *Stream;
```

Description

The **puts** subroutine writes the string pointed to by the *String* parameter to the standard output stream, **stdout**, and appends a new-line character to the output.

The **fputs** subroutine writes the null-terminated string pointed to by the *String* parameter to the output stream specified by the *Stream* parameter. The **fputs** subroutine does not append a new-line character.

Neither subroutine writes the terminating null character.

After the **fputwc**, **putwc**, **fputc**, **fputs**, **puts**, or **putw** subroutine runs successfully, and before the next successful completion of a call either to the **fflush** or **fclose** subroutine on the same stream or a call to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Parameters

Item	Description
String	Points to a string to be written to output.
Stream	Points to the **FILE** structure of an open file.

Return Values

Upon successful completion, the **puts** and **fputs** subroutines return the number of characters written. Otherwise, both subroutines return **EOF**, set an error indicator for the stream and set the **errno** global variable to indicate the error. This happens if the routines try to write to a file that has not been opened for writing.

Error Codes

If the **puts** or **fputs** subroutine is unsuccessful because the output stream specified by the *Stream* parameter is unbuffered or the buffer needs to be flushed, it returns one or more of the following error codes:

Item	Description
EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor specified by the *Stream* parameter and the process would be delayed in the write operation.
EBADF	Indicates that the file descriptor specified by the *Stream* parameter is not a valid file descriptor open for writing.
EFBIG	Indicates that an attempt was made to write to a file that exceeds the process' file size limit or the systemwide maximum file size.
EINTR	Indicates that the write operation was terminated due to receipt of a signal and no data was transferred. **Note:** Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** subroutine regarding the **SA_RESTART** bit.
EIO	Indicates that the process is a member of a background process group attempting to perform a write to its controlling terminal, the **TOSTOP** flag is set, the process is neither ignoring or blocking the **SIGTTOU** signal, and the process group of the process has no parent process.
ENOSPC	Indicates that there was no free space remaining on the device containing the file specified by the *Stream* parameter.

Item	Description
EPIPE	Indicates that an attempt is made to write to a pipe or first-in-first-out (FIFO) that is not open for reading by any process. A **SIGPIPE** signal will also be sent to the process.
ENOMEM	Indicates that insufficient storage space is available.
ENXIO	Indicates that a request was made of a nonexistent device, or the request was outside the capabilities of the device.

putuserattrs Subroutine

Purpose

Stores multiple user attributes in the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int putuserattrs (User, Attributes, Count)
char * User;
dbattr_t * Attributes;
int Count
```

Description

The **putuserattrs** subroutine writes multiple user attributes into the user database. If the database is not already open, this subroutine does an implicit open for reading and writing. Data changed by **putuserattrs** must be explicitly committed by calling the **putuserattr** subroutine with a *Type* parameter specifying the **SEC_COMMIT** value. Until the data is committed, only **get** subroutine calls within the process return the written data.

The *Attributes* array contains information about each attribute that is to be written. Each element in the *Attributes* array must be examined upon a successful call to **putuserattrs** to determine if the *Attributes* array entry was successfully put. Please see **putuserattr** man page for the supported attributes. The **dbattr_t** data structure contains the following fields:

attr_name
The name of the desired attribute.

attr_idx
Used internally by the **putuserattrs** subroutine.

attr_type
The type of the desired attribute. The list of attribute types is defined in the **usersec.h** header file.

S_DOMAINS
The domains for the user. It can be one or more. The attribute type is **SEC_LIST**.

attr_flag
The results of the request to write the desired attribute.

attr_un
A union containing the returned values. Its union members that follow correspond to the definitions of the **attr_char**, **attr_int**, **attr_long**, and **attr_llong** macros, respectively:

au_char
Attributes of type **SEC_CHAR** and **SEC_LIST** contain a pointer to the value to be written.

au_int
Attributes of type **SEC_INT** and **SEC_BOOL** contain the value of the attribute to be written.

au_long
Attributes of type **SEC_LONG** contain the value of the attribute to be written.

au_llong
Attributes of type **SEC_LLONG** contain the value of the attribute to be written.

attr_domain
The authentication domain containing the attribute. The **putuserattrs** subroutine stores the name of the authentication domain that was used to write this attribute if it is not initialized by the caller. The **putuserattrs** subroutine is responsible for managing the memory referenced by this pointer. If **attr_domain** is specified for an attribute, the put request is sent only to that domain. If **attr_domain** is not specified (that is, set to NULL), **putuserattrs** attempts to put the attributes to the first domain associated with the user. All put requests for the attributes with a NULL **attr_domain** are sent to the same domain. In other words, values cannot be put into different domains where **attr_domain** is unspecified; **attr_domain** is set to the name of the domain where the value is put and returned to the invoker. When **attr_domain** is not specified, the list of searchable domains can be restricted to a particular domain by using the **setauthdb** function call.

Use the **setuserdb** and **enduserdb** subroutines to open and close the user database. Failure to explicitly open and close the user database can result in loss of memory and performance.

Parameters

Item	Description
User	Specifies the name of the user for which the attributes are to be written.
Attributes	A pointer to an array of one or more elements of type **dbattr_t**. The list of user attributes is defined in the **usersec.h** header file.
Count	The number of array elements in *Attributes*.

Security

Files accessed:

Item	Description
Mode	**File**
rw	**/etc/group**
rw	**/etc/passwd**
rw	**/etc/security/audit/config**
rw	**/etc/security/environ**
rw	**/etc/security/group**
rw	**/etc/security/lastlog**
rw	**/etc/security/limits**
rw	**/etc/security/passwd**
rw	**/etc/security/pwdhist.dir**
rw	**/etc/security/pwdhist.pag**
rw	**/etc/security/smitacl.user**

Item	Description
rw	**/etc/security/user.roles**

Return Values

The **putuserattrs** subroutine returns a value of 0 if the *User* exists, even in the case when no attributes in the *Attributes* array were successfully updated. Otherwise, a value of -1 is returned and the `errno` global variable is set to indicate the error.

Error Codes

The **putuserattrs** subroutine fails if one or more of the following is true:

Item	Description
EACCES	The system information database could not be accessed for writing.
EINVAL	The *User* parameter is the NULL pointer.
EINVAL	The *Attributes* parameter does not point to valid data for the requested attribute. Limited testing is possible and all errors might not be detected.
EINVAL	The *Attributes* parameter does not point to valid data for the requested attribute. Limited testing is possible and all errors might not be detected.
ENOENT	The specified *User* parameter does not exist.

If the **putuserattrs** subroutine fails to write an attribute, one or more of the following errors is returned in the **attr_flag** field of the corresponding *Attributes* element:

Item	Description
EACCES	The user does not have access to the attribute specified in the *attr_name* field.
EINVAL	The **attr_type** field in the *Attributes* entry contains an invalid type.
EINVAL	The **attr_un** field in the *Attributes* entry does not point to a valid buffer or to valid data for this type of attribute. Limited testing is possible and all errors might not be detected.
ENOATTR	The **attr_name** field in the *Attributes* entry specifies an attribute that is not defined for this user.

Examples

The following sample test program displays the output to a call to **putuserattrs**. In this example, the system has a user named `foo`.

```
#include <stdio.h>
#include <strings.h>
#include <string.h>
#include <usersec.h>


char * CommaToNSL(char *);

#define NATTR         4       /* Number of attributes to be put */
#define USERNAME    "foo"    /* User name */
#define DOMAIN        "files"  /* domain where attributes are going to put. */

main(int argc, char *argv[]) {
    int      rc;
    int      i;
    dbattr_t attributes[NATTR];

        /* Open the user database */
        setuserdb(S_WRITE);
```

```
    /* Valid put */

    attributes[0].attr_name = S_GECOS;
        attributes[0].attr_type = SEC_CHAR;
    attributes[0].attr_domain = DOMAIN;
    attributes[0].attr_char = strdup("I am foo");

    /* Invalid put */

    attributes[1].attr_name = S_LOGINCHK;
        attributes[1].attr_type = SEC_BOOL;
    attributes[1].attr_domain = DOMAIN;
    attributes[1].attr_char = strdup("allow");

    /* Valid put */

    attributes[2].attr_name = S_MAXAGE;
    attributes[2].attr_type = SEC_INT;
    attributes[2].attr_domain = DOMAIN;
    attributes[2].attr_int = 10;

    /* Valid put */

    attributes[3].attr_name = S_GROUPS;
    attributes[3].attr_type = SEC_LIST;
    attributes[3].attr_domain = DOMAIN;
    attributes[3].attr_char = CommaToNSL("staff,system");

    rc = putuserattrs(USERNAME, attributes, NATTR);

    if (rc) {
        printf("putuserattrs failed \n");
        goto clean_exit;
    }

    for (i = 0; i < NATTR; i++) {
        if (attributes[i].attr_flag)
            printf("Put failed for attribute %s. errno = %d \n",
                attributes[i].attr_name, attributes[i].attr_flag);
        else
            printf("Put succeded for attribute %s \n",
                attributes[i].attr_name);
    }

clean_exit:
        enduserdb();

    if (attributes[0].attr_char)
                free(attributes[0].attr_char);

        if (attributes[1].attr_char)
                free(attributes[1].attr_char);

        if (attributes[3].attr_char)
                free(attributes[3].attr_char);

    exit(rc);
}

/*
 * Returns a new NSL created from a comma separated list.
 * The comma separated list is unmodified.
 *
 */
char *
CommaToNSL(char *CommaList)
{
        char    *NSL = (char *) NULL;
        char    *s;

        if (!CommaList)
            return(NSL);

        if (!(NSL = (char *) malloc(strlen(CommaList) + 2)))
            return(NSL);

        strcpy(NSL, CommaList);

        for (s = NSL; *s; s++)
                if (*s == ',')
                        *s = '\0';
```

```
        *(++s) = '\0';
}
```

The following output for the call is expected:

```
Put succeeded for attribute gecos
Put failed for attribute login (errno = 22)
Put succeeded for attribute maxage
Put succeeded for attribute groups
```

putuserpwx Subroutine

Purpose

Accesses the user authentication data.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpw.h>
```

int putuserpwx (*Password***)**
struct userpwx *Password;

Description

The **putuserpwx** subroutine modifies user authentication information. It can be used with those administrative domains that support modifying the user's encrypted password with the **putuserattrs** subroutine. The **chpassx** subroutine must be used to modify authentication information for administrative domains that do not support that functionality.

The **putuserpwx** subroutine updates or creates password authentication data for the user defined in the *Password* parameter in the administrative domain that is specified. The password entry created by the **putuserpwx** subroutine is used only if there is an ! (exclamation point) in the user's password (**S_PWD**) attribute. The user application can use the **putuserattrs** subroutine to add an ! to this field.

The **putuserpwx** subroutine opens the authentication database read-write if no other access has taken place, but the program should call **setpwdb** (**S_READ** | **S_WRITE**) before calling the **putuserpwx** subroutine and **endpwdb** when access to the authentication information is no longer required.

The administrative domain specified in the **upw_authdb** field is set by the **getuserpwx** subroutine. It must be specified by the application program if the **getuserpwx** subroutine is not used to produce the *Password* parameter.

Parameters

Item	Description
Password	Specifies the password structure used to update the password information for this user. The fields in a **userpwx** structure are defined in the **userpw.h** file and contains the following members: **upw_name** Specifies the user's name. **upw_passwd** Specifies the user's encrypted password. **upw_lastupdate** Specifies the time, in seconds, since the epoch (that is, 00:00:00 GMT, 1 January 1970), when the password was last updated. **upw_flags** Specifies attributes of the password. This member is a bit mask of one or more of the following values, defined in the **userpw.h** file: **PW_NOCHECK** Specifies that new passwords need not meet password restrictions in effect for the system. **PW_ADMCHG** Specifies that the password was last set by an administrator and must be changed at the next successful use of the login or **su** command. **PW_ADMIN** Specifies that password information for this user can only be changed by the root user. **upw_authdb** Specifies the administrative domain containing the authentication data.

Security

Files accessed:

Item	Description
Mode	**File**
rw	**/etc/security/passwd**

Return Values

If successful, the **putuserpwx** subroutine returns a value of 0. If the subroutine failed to update or create the password information, the **putuserpwx** subroutine returns a nonzero value.

Error Codes

The **getuserpwx** subroutine fails if the following value is true:

Item	Description
ENOENT	The user does not have an entry in the **/etc/security/passwd** file.

Subroutines invoked by the **putuserpwx** subroutine can also set errors.

Files

Item	Description
/etc/security/passwd	Contains user passwords.

putwc, putwchar, or fputwc Subroutine

Purpose

Writes a character or a word to a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

wint_t putwc(*Character*, *Stream***)**
wint_t *Character*;
FILE ***Stream**;

```
wint_t putwchar(Character)
wint_t Character;
```

```
wint_t fputwc(Character, Stream)
wint_t Character;
FILE Stream;
```

Description

The **putwc** subroutine writes the wide character specified by the *Character* parameter to the output stream pointed to by the *Stream* parameter. The wide character is written as a multibyte character at the associated file position indicator for the stream, if defined. The subroutine then advances the indicator. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

The **putwchar** subroutine works like the **putwc** subroutine, except that **putwchar** writes the specified wide character to the standard output.

The **fputwc** subroutine works the same as the **putwc** subroutine.

Output streams, with the exception of **stderr**, are buffered by default if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen** subroutine causes it to become buffered or line-buffered. Use the **setbuf** subroutine to change the stream's buffering strategy.

After the **fputwc**, **putwc**, **fputc**. **putc**, **fputs**, **puts**, or **putw** subroutine runs successfully, and before the next successful completion of a call either to the **fflush** or **fclose** subroutine on the same stream or to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Parameters

Item	Description
Character	Specifies a wide character of type **wint_t**.

Item	Description
Stream	Specifies a stream of output data.

Return Values

Upon successful completion, the **putwc**, **putwchar**, and **fputwc** subroutines return the wide character that is written. Otherwise **WEOF** is returned, the error indicator for the stream is set, and the **errno** global variable is set to indicate the error.

Error Codes

If the **putwc**, **putwchar**, or **fputwc** subroutine fails because the stream is not buffered or data in the buffer needs to be written, it returns one or more of the following error codes:

Item	Description
EAGAIN	Indicates that the **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter, delaying the process during the write operation.
EBADF	Indicates that the file descriptor underlying the *Stream* parameter is not valid and cannot be updated during the write operation.
EFBIG	Indicates that the process attempted to write to a file that already equals or exceeds the file-size limit for the process. The file is a regular file and an attempt was made to write at or beyond the offset maximum associated with the corresponding stream.
EILSEQ	Indicates that the wide-character code does not correspond to a valid character.
EINTR	Indicates that the process has received a signal that terminates the read operation.
EIO	Indicates that the process is in a background process group attempting to perform a write operation to its controlling terminal. The **TOSTOP** flag is set, the process is not ignoring or blocking the **SIGTTOU** flag, and the process group of the process is orphaned.
ENOMEM	Insufficient storage space is available.
ENOSPC	Indicates that no free space remains on the device containing the file.
ENXIO	Indicates a request was made of a non-existent device, or the request was outside the capabilities of the device.
EPIPE	Indicates that the process has attempted to write to a pipe or first-in-first-out (FIFO) that is not open for reading. The process will also receive a **SIGPIPE** signal.

putws or fputws Subroutine

Purpose

Writes a wide-character string to a stream.

Library

Standard I/O Library (**libc.a**)

Syntax

```
#include <stdio.h>

int putws ( String)
const wchar_t *String;
```

```
int fputws (String,  Stream)
const wchar_t *String;
FILE *Stream;
```

Description

The **putws** subroutine writes the **const wchar_t** string pointed to by the *String* parameter to the standard output stream (**stdout**) as a multibyte character string and appends a new-line character to the output. In all other respects, the **putws** subroutine functions like the **puts** subroutine.

The **fputws** subroutine writes the **const wchar_t** string pointed to by the *String* parameter to the output stream as a multibyte character string. In all other respects, the **fputws** subroutine functions like the **fputs** subroutine.

After the **putws** or **fputws** subroutine runs successfully, and before the next successful completion of a call to the **fflush** or **fclose** subroutine on the same stream or a call to the **exit** or **abort** subroutine, the `st_ctime` and `st_mtime` fields of the file are marked for update.

Parameters

Item	Description
String	Points to a string to be written to output.
Stream	Points to the **FILE** structure of an open file.

Return Values

Upon successful completion, the **putws** and **fputws** subroutines return a nonnegative number. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **putws** or **fputws** subroutine is unsuccessful if the stream is not buffered or data in the buffer needs to be written, and one of the following errors occur:

Item	Description
EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying the *Stream* parameter, which delays the process during the write operation.
EBADF	The file descriptor underlying the *Stream* parameter is not valid and cannot be updated during the write operation.
EFBIG	The process attempted to write to a file that already equals or exceeds the file-size limit for the process.
EINTR	The process has received a signal that terminates the read operation.
EIO	The process is in a background process group attempting to perform a write operation to its controlling terminal. The **TOSTOP** flag is set, the process is not ignoring or blocking the **SIGTTOU** flag, and the process group of the process is orphaned.
ENOSPC	No free space remains on the device containing the file.
EPIPE	The process has attempted to write to a pipe or first-in-first-out (FIFO) that is not open for reading. The process also receives a **SIGPIPE** signal.
EILSEQ	The **wc** wide-character code does not correspond to a valid character.

pwdrestrict_method Subroutine

Purpose

Defines loadable password restriction methods.

Library

Syntax

```
int pwdrestrict_method (UserName, NewPassword, OldPassword, Message)
char * UserName;
char * NewPassword;
char * OldPassword;
char ** Message;
```

Description

The **pwdrestrict_method** subroutine extends the capability of the password restrictions software and lets an administrator enforce password restrictions that are not provided by the system software.

Whenever users change their passwords, the system software scans the **pwdchecks** attribute defined for that user for site specific restrictions. Since this attribute field can contain load module file names, for example, methods, it is possible for the administrator to write and install code that enforces site specific password restrictions.

The system evaluates the **pwdchecks** attribute's value field in a left to right order. For each method that the system encounters, the system loads and invokes that method. The system uses the **load** subroutine to load methods. It invokes the **load** subroutine with a *Flags* value of **1** and a *LibraryPath* value of **/usr/lib**. Once the method is loaded, the system invokes the method.

To create a loadable module, use the **-e** flag of the **ld** command. Note that the name **pwdrestrict_method** given in the syntax is a generic name. The actual subroutine name can be anything (within the compiler's name space) except **main**. What is important is, that for whatever name you choose, you must inform the **ld** command of the name so that the **load** subroutine uses that name as the entry point into the module. In the following example, the C compiler compiles the **pwdrestrict.c** file and pass **-e pwdrestrict_method** to the **ld** command to create the method called **pwdrestrict**:

```
cc -e pwdrestrict_method -o pwdrestrict pwdrestrict.c
```

The convention of all password restriction methods is to pass back messages to the invoking subroutine. Do not print messages to stdout or stderr. This feature allows the password restrictions software to work across network connections where stdout and stderr are not valid. Note that messages must be returned in dynamically allocated memory to the invoking program. The invoking program will deallocate the memory once it is done with the memory.

There are many caveats that go along with loadable subroutine modules:

1. The values for *NewPassword* and *OldPassword* are the actual clear text passwords typed in by the user. If you copy these passwords into other parts of memory, clear those memory locations before returning back to the invoking program. This helps to prevent clear text passwords from showing up in core dumps. Also, do not copy these passwords into a file or anywhere else that another program can access. Clear text passwords should never exist outside of the process space.
2. Do not modify the current settings of the process' signal handlers.
3. Do not call any functions that will terminate the execution of the program (for example, the **exit** subroutine, the **exec** subroutine). Always return to the invoking program.
4. The code must be thread-safe.

5. The actual load module must be kept in a write protected environment. The load module and directory should be writable only by the root user.

One last note, all standard password restrictions are performed before any of the site specific methods are invoked. Thus, methods are the last restrictions to be enforced by the system.

Parameters

Item	Description
UserName	Specifies a "local" user name.
NewPassword	Specifies the new password in clear text (not encrypted).This value may be a NULL pointer. Clear text passwords are always in 7 bit ASCII.
OldPassword	Specifies the current password in clear text (not encrypted).This value may be a NULL pointer. Clear text passwords are always in 7 bit ASCII.
Message	Specifies the address of a pointer to **malloc**'ed memory containing an NLS error message. The method is expected to supply the **malloc**'ed memory and the message.

Return Values

The method is expected to return the following values. The return values are listed in order of precedence.

Item	Description
-1	Internal error. The method could not perform its password evaluation. The method must set the **errno** variable. The method must supply an error message in *Message* unless it can't allocate memory for the message. If it cannot allocate memory, then it must return the NULL pointer in *Message*.
1	Failure. The password change did not meet the requirements of the restriction. The password restriction was properly evaluated and the password change was not accepted. The method must supply an error message in *Message*. The **errno** variable is ignored. Note that composition failures are cumulative, thus, even though a failure condition is returned, trailing composition methods will be invoked.
0	Success. The password change met the requirements of the restriction. If necessary, the method may supply a message in *Message*; otherwise, return the NULL pointer. The **errno** variable is ignored.

q

The following Base Operating System (BOS) runtime services begin with the letter *q*.

quantized32, quantized64, or quantized128 Subroutine

Purpose

Sets the exponent of the first parameter to the exponent of the second parameter, attempting to keep the value the same.

Syntax

```
#include <math.h>

_Decimal32 quantized32 (x, y)
_Decimal32 x;
_Decimal32 y;

_Decimal64 quantized64 (x, y)
_Decimal64 x;
_Decimal64 y;

_Decimal128 quantized128 (x, y)
_Decimal128 x;
_Decimal128 y;
```

Description

The **quantized32**, **quantized64**, and **quantized128** subroutines set the exponent of the *x* parameter to the exponent of *y* parameter, while attempting to keep the value of the *x* parameter the same. If the exponent is increased, the value is correctly rounded according to the current rounding mode; if the result does not have the same value as that of the *x* parameter, the inexact floating-point exception is raised. If the exponent is decreased and the significand of the result has more digits than the type allows, the result is NaN and the **invalid** floating-point exception is raised.

If one or both of the operands are NaN, the result is NaN. If only one operand is infinite, the result is NaN and the **invalid** floating-point exception is raised. If both operands are infinite, the result is DEC_INFINITY and the sign is the same as that of the *x* parameter.

An application checking for error situations should set the value of the **errno** global variable to zero and call the **feclearexcept** (**FE_ALL_EXCEPT**) subroutine before calling these subroutines. Upon return, if the value of the **errno** global variable is nonzero or the return value of the **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) subroutine is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.
y	Specifies the value to be computed.

Return Values

The **quantized32**, **quantized64**, and **quantized128** subroutines return the number that is equal to the *x* parameter in value (except for any rounding) and sign and has an exponent equal to that of the *y* parameter.

quick_exit Subroutine

Purpose

This subroutine causes normal program termination to occur without completely cleaning the resources.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
_Noreturn void quick_exit(int status);
```

Description

The **quick_exit** subroutine causes normal program termination to occur. Subroutines that are registered by the **atexit** subroutine or signal handlers that are registered by the **signal** subroutine are not called. If a program calls the **quick_exit** subroutine more than one time or if the program calls the exit subroutine in addition to the **quick_exit** subroutine, the behavior is unspecified. If a signal is raised while the **quick_exit** subroutine is running, the behavior is unspecified.

The **quick_exit** subroutine first calls all subroutines that are registered by the **at_quick_exit** subroutine, in the reverse order of their registration, except that a subroutine is called after any previously registered subroutines which are already being called at the time it was registered. If during the call to any such subroutine, a call to the **longjmp** subroutine is made that might stop the call to the registered subroutine, the behavior is undefined.

The control is returned to the host environment by the **_Exit(status)** subroutine call.

Return Values

The **quick_exit** cannot return any value to its caller.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

qsort Subroutine

Purpose

Sorts a table of data in place.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
void qsort (Base, NumberOfElements, Size, ComparisonPointer)
```

```
void * Base;
size_t  NumberOfElements,  Size;
int  (*(*ComparisonPointer))(const void*, const void*);
```

Description

The **qsort** subroutine sorts a table of data in place. It uses the quicker-sort algorithm.

Parameters

Item	Description
Base	Points to the element at the base of the table.
NumberOfElements	Specifies the number of elements in the table.
Size	Specifies the size of each element.
ComparisonPointer	Points to the comparison function, which is passed two parameters that point to the objects being compared. The **qsort** subroutine sorts the array in ascending order according to the comparison function.

Return Values

The comparison function compares its parameters and returns a value as follows:

- If the first parameter is less than the second parameter, the *ComparisonPointer* parameter returns a value less than 0.
- If the first parameter is equal to the second parameter, the *ComparisonPointer* parameter returns 0.
- If the first parameter is greater than the second parameter, the *ComparisonPointer* parameter returns a value greater than 0.

Because the comparison function need not compare every byte, the elements can contain arbitrary data in addition to the values being compared.

Note: If two items are the same when compared, their order in the output of this subroutine is unpredictable.

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

quotactl Subroutine

Purpose

Manipulates disk quotas.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/fs/quota_common.h>
int quotactl (Path, Cmd, ID, Addr)
int  Cmd, ID;
char * Addr, * Path;
```

Description

The **quotactl** subroutine enables, disables, and manipulates disk quotas for file systems on which quotas have been enabled.

On AIX, disk quotas are supported by the legacy Journaled File System (JFS) and the enhanced Journaled File System (JFS2).

The *Cmd* parameter is constructed through use of the **QCMD(***Qcmd***,** *type***)** macro contained within the **sys/fs/quota_common.h** file. The *Qcmd* parameter specifies the quota control command. The *type* parameter specifies either user (**USRQUOTA**) or group (**GRPQUOTA**) quota type.

The valid values for the *Cmd* parameter in all supported file system types are:

Q_QUOTAON
Enables disk quotas for the file system specified by the *Path* parameter. The *Addr* parameter specifies a file from which to take the quotas. The quota file must exist; it is normally created with the **quotacheck** command. The *ID* parameter is unused. Root user authority is required to enable quotas. By specifying the new quota file path in the *Addr* parameter, the **quotactl** command can also be used to change the quota file that is being used without first disabling disk quotas.

Q_QUOTAOFF
Disables disk quotas for the file system specified by the *Path* parameter. The *Addr* and *ID* arguments are unused. Root user authority is required to disable quotas.

Additional JFS specific values for the *Cmd* parameter are as follows:

Q_GETQUOTA
Gets disk quota limits and current usage for a user or group specified by the *ID* parameter. The *Addr* parameter points to a **dqblk** buffer to hold the returned information. The **dqblk** structure is defined in the **jfs/quota.h** file. Root user authority is required if the *ID* value is not the current ID of the caller.

Q_SETQUOTA
Sets disk quota limits for the user or group specified by the *ID* parameter. The *Addr* parameter points to a **dqblk** buffer containing the new quota limits. The **dqblk** structure is defined in the **jfs/quota.h** file. Root user authority is required to set quotas.

Q_SETUSE
Sets disk usage limits for the user or group specified by the *ID* parameter. The *Addr* parameter points to a **dqblk** buffer containing the new usage limits. The **dqblk** structure is defined in the **jfs/quota.h** file. Root user authority is required to set disk usage limits.

Additional JFS2 specific values for the *Cmd* parameter are as follows:

Q_J2GETQUOTA
Gets quota limits, current usage, and time remaining in grace periods for the user or group specified by the *ID* parameter. The *Addr* parameter points to a **quota64_t** buffer to hold the returned information. The **quota64_t** structure is defined in the **quota_common.h** file. Root user authority is required if the *ID* value is not the current ID of the caller.

Q_J2PUTQUOTA
Updates (replaces) the current usage values for the user or group specified by the *ID* parameter. The *Addr* parameter points to a **quota64_t** buffer holding the new information. The **quota64_t** structure is defined in the **quota_common.h** file. Root user authority is required.

Q_J2GETLIMIT
Gets quota limits information for the Limits Class specified by the *ID* parameter. The *Addr* parameter points to a **j2qlimit_t** buffer to hold the returned information. The **j2qlimit_t** structure is defined in the **j2/j2_quota.h** file. Root user authority is required.

Q_J2PUTLIMIT
Updates quota limits information for the Limits Class specified by the *ID* parameter. The *Addr* parameter points to a **j2qlimit_t** buffer holding the new information. The **j2qlimit_t** structure is defined in the **j2/j2_quota.h** file. Root user authority is required.

Q_J2NEWLIMIT
Creates a new Limits Class and updates it with the quota limits information from *Addr*. The *ID* parameter is ignored. The *Addr* parameter points to a **j2qlimit_t** buffer holding the new information. The **j2qlimit_t** structure is updated with the new Limits Class ID and returned to the user. The **j2qlimit_t** structure is defined in the **j2/j2_quota.h** file. Root user authority is required.

Q_J2RMVLIMIT
Marks the Limits Class specified by the *ID* parameter as deleted. Any Usage record referencing a deleted Limits Class is now limited by the default Limits Class. The *Addr* parameter is ignored. Root user authority is required.

Q_J2DEFLIMIT
Sets the Limits Class specified by the *ID* parameter as the default Limits Class. The *Addr* parameter is ignored. Root user authority is required.

Q_J2USELIMIT
Binds a Usage record to the Limits Class specified by the *ID* parameter. The Limits Class must be valid; otherwise, **ENOENT** is returned. Use the *Addr* parameter to pass a pointer to the user ID or group ID. Root user authority is required.

Q_J2GETNEXTQ
Returns the ID of the next allocated, nondeleted Limits Class higher than the ID specified by the *ID* parameter. The *Addr* parameter points to a buffer containing a **uid_t** structure. Root user authority is required.

Q_J2INITFILE
Initializes an existing quota file. The *Addr* and *ID* parameters are ignored. Root user authority is required.

Q_J2QUOTACHK
Performs a consistency check on an existing quota file. If any of the control data within the file is invalid or inconsistent, **Q_J2QUOTACHK** attempts to reconstruct the control data based on existing quota data in the file. If no **qwuota** data can be recognized, the file is initialized. The *Addr* and *ID* parameters are ignored. Root user authority is required.

Q_J2DELQUOTA
Deletes the passed-in users or groups if there are no files owned by them. The space is returned to the quota file free list so it can be reused. The *Addr* parameter points to an array of **qid_t** elements, with at most **MAXDELIDS** elements. The *ID* parameter contains the count of the elements in the array. The **qid_t** type is defined in the **j2/j2_quota.h** file and the **MAXDELIDS** is defined in the **sys/fs/quota_common.h** file. Root user authority is required to delete quotas.

Parameters

Item	Description
Path	Specifies the path name of any file within the mounted file system to which the quota control command is to be applied. Typically, this would be the mount point of the file system.
Cmd	Specifies the quota control command to be applied and whether it is applied to a user or group quota.
ID	Specifies the user or group ID to which the quota control command applies. The *ID* parameter is interpreted by the specified quota type. The JFS file system supports quotas for IDs within the range of **MINDQUID** through **MAXDQID**; JFS2 supports all IDs.
Addr	Points to the address of an optional, command-specific, data structure that is copied in or out of the system. The interpretation of the *Addr* parameter for each quota control command is given earlier.

Return Values

A successful call returns 0; otherwise, the value -1 is returned and the **errno** global variable indicates the reason for the failure.

Error Codes

A **quotactl** subroutine will fail when one of the following occurs:

Item	Description
EACCES	In the **Q_QUOTAON** command, the quota file is not a regular file.
EACCES	Search permission is denied for a component of a path prefix.
EFAULT	An invalid *Addr* parameter is supplied; the associated structure could not be copied in or out of the kernel.
EFAULT	The *Path* parameter points outside the process's allocated address space.
EINVAL	The specified quota control command or quota type is invalid.
EINVAL	Path name contains a character with the high-order bit set.
EINVAL	The *ID* parameter is outside of the supported range of **MINDQID** through **MAXDQID** (JFS only).
EINVAL	The *ID* parameter is negative or larger than **MAXDELIDS** when deleting quota entries (JFS2 only).
EIO	An I/O error occurred while reading or writing the quotas file.
ELOOP	Too many symbolic links were encountered in translating a path name.
ENAMETOOLONG	A component of either path name exceeded 255 characters, or the entire length of either path name exceeded 1023 characters.
ENOENT	A file name does not exist.
ENOTBLK	Mounted file system is not a block device.
ENOTDIR	A component of a path prefix is not a directory.
EOPNOTSUPP	The file system does not support quotas.
EPERM	The quota control commands is privileged and the caller did not have root user authority.
EROFS	In the **Q_QUOTAON** command, the quota file resides on a read-only file system.
EUSERS	The in-core quota table cannot be expanded (JFS only).
ENOMEM	Unable to allocate memory.

r

The following Base Operating System (BOS) runtime services begin with the letter *r*.

raise Subroutine

Purpose

Sends a signal to the currently running program.

Libraries

Standard C Library (**libc.a**)

Threads Library (**libpthreads.a**)

Syntax

```
#include <sys/signal.h>
```

```
int raise ( Signal)
int Signal;
```

Description

The **raise** subroutine sends the signal specified by the *Signal* parameter to the executing process or thread, depending if the POSIX threads API (the **libpthreads.a** library) is used or not. When the program is not linked with the threads library, the **raise** subroutine sends the signal to the calling process as follows:

```
return kill(getpid(), Signal);
```

When the program is linked with the threads library, the **raise** subroutine sends the signal to the calling thread as follows:

```
return pthread_kill(pthread_self(), Signal);
```

When using the threads library, it is important to ensure that the threads library is linked before the standard C library.

Parameter

Item	Description
Signal	Specifies a signal number.

Return Values

Upon successful completion of the **raise** subroutine, a value of 0 is returned. Otherwise, a nonzero value is returned, and the **errno** global variable is set to indicate the error.

Error Code

Item	Description
EINVAL	The value of the sig argument is an invalid signal number

rand or srand Subroutine

Purpose

Generates pseudo-random numbers.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
int rand
```

void srand (*Seed*)
unsigned int *Seed*;

Description

Attention: Do not use the **rand** subroutine in a multithreaded environment. See the multithread alternative in the **rand_r** (“rand_r Subroutine” on page 1683) subroutine article.

The **rand** subroutine generates a pseudo-random number using a multiplicative congruential algorithm. The random-number generator has a period of 2**32, and it returns successive pseudo-random numbers in the range from 0 through (2**15) -1.

The **srand** subroutine resets the random-number generator to a new starting point. It uses the *Seed* parameter as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to the **rand** subroutine. If you then call the **srand** subroutine with the same seed value, the **rand** subroutine repeats the sequence of pseudo-random numbers. When you call the **rand** subroutine before making any calls to the **srand** subroutine, it generates the same sequence of numbers that it would if you first called the **srand** subroutine with a seed value of 1.

Note: The **rand** subroutine is a simple random-number generator. Its spectral properties, a mathematical measurement of randomness, are somewhat limited. See the **drand48** subroutine or the **random** subroutine for more elaborate random-number generators that have greater spectral properties.

Parameter

Item	Description
Seed	Specifies an initial seed value.

Return Values

Upon successful completion, the **rand** subroutine returns the next random number in sequence. The **srand** subroutine returns no value.

There are better random number generators, as noted above; however, the **rand** and **srand** subroutines are the interfaces defined for the ANSI C library.

Example

The following functions define the semantics of the **rand** and **srand** subroutines, and are included here to facilitate porting applications from different implementations:

```
static unsigned int next = 1;
int rand( )
{
next = next
*
 1103515245 + 12345;
return ((next >>16) & 32767);
}
```

```
void srand (Seed)


unsigned
int Seed;
{
next = Seed;
}
```

rand_r Subroutine

Purpose

Generates pseudo-random numbers.

Libraries

Thread-Safe C Library (**libc_r.a**)

Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <stdlib.h>
```

int rand_r (*Seed***)**
unsigned int * *Seed*;

Description

The **rand_r** subroutine generates and returns a pseudo-random number using a multiplicative congruential algorithm. The random-number generator has a period of 2**32, and it returns successive pseudo-random numbers.

Note: The **rand_r** subroutine is a simple random-number generator. Its spectral properties (the mathematical measurement of the randomness of a number sequence) are limited. See the **drand48** subroutine or the **random** (“random, srandom, initstate, or setstate Subroutine” on page 1684) subroutine for more elaborate random-number generators that have greater spectral properties.

Programs using this subroutine must link to the **libpthreads.a** library.

Parameter

Item	Description
Seed	Specifies an initial seed value.

Return Values

Item	Description
0	Indicates that the subroutines was successful.
-1	Indicates that the subroutines was not successful.

Error Codes

If the following condition occurs, the **rand_r** subroutine sets the **errno** global variable to the corresponding value.

Item	Description
EINVAL	The *Seed* parameter specifies a null value.

File

Item	Description
/usr/include/sys/types.h	Defines system macros, data types, and subroutines.

random, srandom, initstate, or setstate Subroutine

Purpose

Generates pseudo-random numbers more efficiently.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
long random ( )
```

```
void srandom (Seed)
unsigned int Seed;
```

```
char *initstate ( Seed,  State,  Number)
unsigned int Seed;
char *State;
size_t Number;
```

```
char *setstate (State)
const char *State;
```

Description

Attention: Do not use the **random**, **srandom**, **initstate**, or **setstate** subroutine in a multithreaded environment.

The **random** subroutine uses a non-linear additive feedback random-number generator employing a default-state array size of 31 long integers to return successive pseudo-random numbers in the range from 0 to 2**31-1. The period of this random number generator is very large, approximately 16 *

(2**31-1). The size of the state array determines the period of the random number generator. Increasing the state array size increases the period.

With a full 256 bytes of state information, the period of the random-number generator is greater than 2**69, which should be sufficient for most purposes.

The **random** and **srandom** subroutines have almost the same calling sequence and initialization properties as the **rand** and **srand** subroutines. The difference is that the **rand** subroutine produces a much less random sequence; in fact, the low dozen bits generated by the **rand** subroutine go through a cyclic pattern. All the bits generated by the **random** subroutine are usable. For example, `random( )&01` produces a random binary value.

The **srandom** subroutine, unlike the **srand** subroutine, does not return the old seed because the amount of state information used is more than a single word. The **initstate** subroutine and **setstate** subroutine handle restarting and changing random-number generators. Like the **rand** subroutine, however, the **random** subroutine by default produces a sequence of numbers that can be duplicated by calling the **srandom** subroutine with 1 as the seed.

The **initstate** subroutine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by the **initstate** subroutine, to decide how sophisticated a random-number generator it should use; the larger the state array, the more random are the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. For amounts greater than or equal to 8 bytes, or less than 32 bytes, the **random** subroutine uses a simple linear congruential random number generator, while other amounts are rounded down to the nearest known value. The *Seed* parameter specifies a starting point for the random-number sequence and provides for restarting at the same point. The **initstate** subroutine returns a pointer to the previous state information array.

Once a state has been initialized, the **setstate** subroutine allows rapid switching between states. The array defined by *State* parameter is used for further random-number generation until the **initstate** subroutine is called or the **setstate** subroutine is called again. The **setstate** subroutine returns a pointer to the previous state array.

After initialization, a state array can be restarted at a different point in one of two ways:

- The **initstate** subroutine can be used, with the desired seed, state array, and size of the array.
- The **setstate** subroutine, with the desired state, can be used, followed by the **srandom** subroutine with the desired seed. The advantage of using both of these subroutines is that the size of the state array does not have to be saved once it is initialized.

Parameters

Item	Description
Seed	Specifies an initial seed value.
State	Points to the array of state information.
Number	Specifies the size of the state information array.

Error Codes

If the **initstate** subroutine is called with less than 8 bytes of state information, or if the **setstate** subroutine detects that the state information has been damaged, error messages are sent to standard error.

raw or noraw Subroutine

Purpose

Places the terminal into or out of raw mode.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
raw( )
noraw( )
```

Description

The **raw** or **noraw** subroutine places the terminal into or out of raw mode, respectively. RAW mode is similar to CBREAK mode (**cbreak** or **nocbreak**. In RAW mode, the system immediately passes typed characters to the user program. The interrupt, quit, and suspend characters are passed uninterrupted, instead of generating a signal. RAW mode also causes 8-bit input and output.

To get character-at-a-time input without echoing, call the **cbreak** and **noecho** subroutines. Most interactive screen-oriented programs require this sort of input.

Return Values

Item	Description
OK	Indicates the subroutine completed. The **raw** and **noraw** routines always return this value.

Examples

1. To place the terminal into raw mode, use:

   ```
   raw();
   ```

2. To place the terminal out of raw mode, use:

   ```
   noraw();
   ```

ra_attach Subroutine

Purpose

Attaches a work component to a resource.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int ra_attach(rstype1, rsid1, rstype2, rsid2, flags)
rstype_t rstype1, rstype2;
rsid_t rsid1, rsid2;
unsigned int flags;
```

Description

The **ra_attach** subroutine attaches a work component specified by the *rstype1* and *rsid1* parameters to the resource specified by the *rstype2* and *rsid2* parameters.

Parameters

Item **Description**

rstype1 Specifies the type of work component to be attached to the resource specified by *rstype2/rsid2*. The *rstype1* parameter must be one of the following defined in rset.h.

R_PROCESS
Existing process

R_THREAD
Existing kernel thread

R_FILDES
File identified by an open file descriptor

R_SHM
Shared memory segment identified by shared memory ID

R_SUBRANGE
Attachment to a memory range within a work component

rsid1 Specifies the work component associated with the *rstype1* parameter. The *rsid1* parameter must be one of the following:

Process ID (for *rstype1* of R_PROCESS)
Set the rsid_t at_pid field to the desired process ID.

Kernel thread ID (for *rstype1* of R_THREAD)
Set the rsid_t.at_tid field to the desired kernel thread ID.

Open file descriptor (for rstype1 of R_FILDES)
Set the rsid_t at_fd field to the desired file descriptor.

Shared memory segment (for rstype of R_SHM)
Set the rsid_t at_shmid field to the desired shared memory ID.

Pointer to a subrange_t struct (for rstype of R_SUBRANGE)
Set the rsid_t at_subbrange field to the address of a subrange_t struct. Set the subrange_t struct su_offset, su_length, su_rstype, and su_rsid fields. The other fields in the subrange_t struct are ignored. The memory allocation policy is taken from the flags parameter, not the su_policy field.

Set the subrange_t su_rstype field to R_PROCMEM and su_rsid.at_pid field to RS_MYSELF to attach to a memory range in the user process. Set the subrange_t su_offset field to the starting address of the range in the process. Set the subrange_t su_length field to the length of the range in the process.

Note: The subrange_t su_offset and su_length fields must be a multiple of 4 KB. For optimum performance, the fields must be the multiple of the page size backing the memory range. The page size used to back a memory range can be obtained using the **vmgetinfo** subroutine specifying the VM_PAGE_INFO command parameter.

rstype2 Specifies the type of the resource to be attached to the work component. The *rstype2* parameter must be one of the following defined in rset.h.

R_RSET
Resource set attachment

R_SRADID
SRADID attachment

Item	Description
rsid2	Specifies the resource associated with the *rstype2* parameter. The *rsid2* parameter must be one of the following: **Resource set (for *rstype2* of R_RSET)** Set the rsid_t at_rset field to the desired resource set. **SRADID (Scheduler Resource Allocation Domain Identifier for *rstype2* of R_SRADID)** Set the rsid_t at_sradid field to the desired sradid. An SRADID may only be attached to a thread or to a memory range. An at_sradid value of SRADID_ANY may be specified on memory range attachments to indicate a memory affinity preference for all memory in the partition.
flags	Specifies memory allocation and other attachment options: **P_DEFAULT** Default memory allocation policy **P_FIRST_TOUCH** First access memory allocation policy **P_BALANCED** Balanced memory allocation policy **R_MIGRATE_ASYNC** Asynchronously migrate physical memory in the address range (for *rstype1* of R_SHM or R_SUBRANGE) **R_MIGRATE_SYNC** Synchronously migrate physical memory in the address range (for *rstype1* of R_SHM or R_SUBRANGE) **R_ATTACH_STRSET** Process is to be scheduled with a single-threaded policy, only on one hardware thread per physical processor (for *rstype1* of R_PROCESS).

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EINVAL	One of the following occurred: • The *flags* parameter contains an invalid value. • The *rstype1* or *rstype2* parameter contains an invalid type identifier.
ENODEV	One of the following occurred: • The resource set specified by the *rstype2* and *rsid2* parameters does not contain any available processors. • An invalid *rsid2* SRADID is specified.
ENOTSUP	One of the following occurred: • An attempt to attach an SRADID is made and ENHANCED_AFFINITY is disabled. • An attempt to attach an SRADID to a file is made. • An R_SUBRANGE request with su_rstype R_PROCMEM is made and the su_rsid.at_pid field is not RS_MYSELF.
ESRCH	A work component specified by the *rstype1* and *rsid1* parameters does not exist.

Item	Description
EPERM	One of the following occurred: • *rstype2* specified R_RSET and calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege. j • *rstype2* specified R_RSET and calling process has neither root authority nor the same effective user ID as the process identified by the *rstype1* and *rsid1* parameters. • *rstype2* specified R_RSET or R_SRADID and the process or thread work component specified by the *rstype1* and *rsid1* parameters has one or more threads with a bindprocessor binding. • *rstype1* and *rsid1* parameters specified a process and *rstype2* and *rsid2* parameters specified a resource set. The processors in the rset are not included in the process's partition resource set or a thread in the specified process has a resource set attachment that is not a subset of the *rstype1/rsid1* resource set. • *rstype2* specified R_SRADID attachment to a memory range that has a resource set attachment.

ra_attachrset Subroutine

Purpose

Attaches a work component to a resource set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int ra_attachrset (rstype, rsid, rset, flags)
rstype_t rstype;
rsid_t rsid;
rsethandle_t rset;
unsigned int flags;
```

Description

The **ra_attachrset** subroutine attaches a work component specified by the *rstype* and *rsid* parameters to a resource set specified by the *rset* parameter.

The work component is an existing process identified by the process ID or an existing kernel thread identified by the kernel thread ID (tid). A process ID or thread ID value of RS_MYSELF indicates the attachment applies to the current process or the current kernel thread, respectively.

The following conditions must be met to successfully attach a process to a resource set:

- The resource set must contain processors that are available in the system.
- The calling process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling process must either have root authority or the same effective userid as the target process.
- The target process must not contain any threads that have bindprocessor bindings to a processor.
- The resource set must be contained in (be a subset of) the target process' partition resource set.
- The resource set must be a superset of all the threads' *rset* in the target process.
- For R_FILDES *rstype*, the calling process must specify an open file descriptor, and it must have write access to the file, or the calling process' effective userid must be equal to the file owner's userid.
- For R_SHM *rstype*, the calling process' effective userid must be equal to the shared segment's owner.

The following conditions must be met to successfully attach a kernel thread to a resource set:

- The resource set must contain processors that are available in the system.
- The calling process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling process must either have root authority or the same effective userid as the target process.
- The target thread must not have bindprocessor bindings to a processor.
- The resource set must be contained in (be a subset of) the target thread's process effective and partition resource set.

If any of these conditions are not met, the attachment will fail.

Once a process is attached to a resource set, the threads in the process will only run on processors contained in the resource set. Once a kernel thread is attached to a resource set, the threads will only run on processors contained in the resource set.

Dynamic Processor Deallocation and DLPAR may invalidate the processor attachment that is being specified. A program must become DLPAR Aware to resolve this problem.

The *flags* parameter can be set to indicate the policy for using the resources contained in the resource set specified in the *rset* parameter. The only supported scheduling policy is R_ATTACH_STRSET, which is useful only when the processors of the system are running in simultaneous multithreading mode. Processors like the POWER5 support simultaneous multithreading, where each physical processor has two execution engines, called *hardware threads*. Each hardware thread is essentially equivalent to a single processor, and each is identified as a separate processor in a resource set. The R_ATTACH_STRSET flag indicates that the process is to be scheduled with a single-threaded policy; namely, that it should be scheduled on only one hardware thread per physical processor. If this flag is specified, then all of the available processors indicated in the resource set must be of exclusive use (the processor must belong to some exclusive use processor resource set). A new resource set, called an *ST resource set*, is constructed from the specified resource set and attached to the process according to the following rules:

- All offline processors are ignored.
- If all the hardware threads (processors) of a physical processor (when running in simultaneous multithreading mode, there will be more than one active hardware thread per physical processor) are not included in the specified resource set, the other processors of the processor are ignored when constructing the ST resource set.
- Only one processor (hardware thread) resource per physical processor is included in the ST resource set.

Parameters

Item	Description
rstype	Specifies the type of work component to be attached to the resource set specified by the *rset* parameter. The *rstype* parameter must be the following value, defined in **rset.h**: **R_PROCESS** Existing process **R_THREAD** Existing kernel thread **R_FILDES** File identified by an open file descriptor **R_SHM** Shared memory segment identified by shared memory segment ID **R_SUBRANGE** Attachment involves a subrange of the work component

Item	Description
rsid	Identifies the work component to be attached to the resource set specified by the *rset* parameter. The *rsid* parameter must be the following: **Process ID (for *rstype* of R_PROCESS)** Set the *rsid_t at_pid* field to the desired process' process ID. **Kernel thread ID (for *rstype* of R_THREAD)** Set the *rsid_t at_tid* field to the desired kernel thread's thread ID. **Open file descriptor (for *rstype* of R_FILDES)** Set the *rsid_t at_fd* field to the desired file descriptor. **Shared memory segment ID (for *rstype* of R_SHM)** Set the *rsid_t at_shmid* field to the desired shared memory ID. **Pointer to a `subrange_t` struct (for *rstype* of R_SUBRANGE)** Set the `subrange_t` *su_offset*, *su_length*, *su_rstype*, and *su_rsid* fields. The other fields in the `subrange_t` struct are ignored. The memory allocation policy is taken from the *flags* parameter, not the *su_policy* field.
rset	Specifies which work component (specified by the *rstype* and *rsid* parameters) to attach to the resource set.
flags	Specifies either the memory allocation or the scheduling policy for the work component being attached. The *flags* parameter must be the following: **P_DEFAULT** Default memory policy **P_FIRST_TOUCH** First access memory policy **P_BALANCED** Balanced memory policy **R_ATTACH_STRSET** Single-threaded scheduling policy If the *rstype* parameter value is set to R_SUBRANGE, the memory allocation policy is specified in the `subrange_t` *su_policy* field rather than in the *flags* parameter. The R_ATTACH_STRSET value is only applicable if the *rstype* parameter value is set to R_PROCESS. The R_ATTACH_STRSET value indicates that the process is to be scheduled with a single-threaded policy (only on one hardware thread per physical processor).

Return Values

If successful , a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **ra_attachrset** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *flags* parameter contains an invalid value. • The *rstype* parameter contains an invalid type qualifier. • The R_ATTACH_STRSET *flags* parameter is specified and one or more processors in the *rset* parameter are not assigned for exclusive use.

Item	Description
ENODEV	The resource set specified by the *rset* parameter does not contain any available processors, or the R_ATTACH_STRSET *flags* parameter is specified and the constructed ST resource set does not have any available processors.
ESRCH	The process or kernel thread identified by the *rstype* and *rsid* parameters does not exist.
EPERM	One of the following is true: • If the *rstype* is R_PROCESS, either the resource set specified by the *rset* parameter is not included in the partition resource set of the process identified by the *rstype* and *rsid* parameters, or any of the thread's R_THREAD *rset* in this process is not a subset of the resource set specified by the *rset* parameter. • If the *rstype* is R_THREAD, the resource set specified by the *rset* parameter is not included in the target thread's process effective or partition (real) resource set. • The calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege. • The calling process has neither root authority nor the same effective user ID as the process identified by the *rstype* and *rsid* parameters. • The process or thread identified by the *rstype* and *rsid* parameters has one or more threads with a bindprocessor processor binding.

ra_detach Subroutine

Purpose

Detaches a work component from a resource.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int ra_detach(rstype1, rsid1, rstype2, rsid2, flags)
rstype_t rstype1, rstype2;
rsid_t rsid1, rsid2;
unsigned int flags;
```

Description

The **ra_detach** subroutine detaches a work component specified by the *rstype1* and *rsid1* parameters from the resource specified by the *rstype2* and *rsid2* parameters.

Parameters

Item **Description**

rstype1 Specifies the type of work component to be detached from the resource specified by *rstype2/rsid2*. The *rstype1* parameter must be one of the following defined in rset.h.

R_PROCESS
Existing process

R_THREAD
Existing kernel thread

R_FILDES
File identified by an open file descriptor

R_SHM
Shared memory segment identified by the shared memory ID

R_SUBRANGE
Attachment to a memory range within a work component

rsid1 Specifies the work component associated with the *rstype1* parameter. The *rsid1* parameter must be one of the following:

Process ID (for *rstype1* of R_PROCESS)
Set the rsid_t at_pid field to the desired process ID.

Kernel thread ID (for *rstype1* of R_THREAD)
Set the rsid_t.at_tid field to the desired kernel thread ID.

Open file descriptor (for rstype1 of R_FILDES)
Set the rsid_t at_fd field to the desired file descriptor.

Shared memory segment (for rstype of R_SHM)
Set the rsid_t at_shmid field to the desired shared memory ID.

Pointer to a subrange_t struct (for rstype of R_SUBRANGE)
Set the rsid_t at_subbrange field to the address of a subrange_t struct. Set the subrange_t struct su_offset, su_length, su_rstype, and su_rsid fields. The other fields in the subrange_t struct are ignored.

Set the subrange_t su_rstype field to R_PROCMEM and su_rsid.at_pid field to RS_MYSELF to detach from a memory range in the user process. Set the subrange_t su_offset field to the starting address of the range in the process. Set the subrange_t su_length field to the length of the range in the process.

Note: The subrange_t su_offset and su_length fields must be a multiple of 4 KB. For optimum performance, the fields must be the multiple of the page size backing the memory range. The page size used to back a memory range can be obtained using the **vmgetinfo** subroutine specifying the VM_PAGE_INFO command parameter.

rstype2 Specifies the type of the resource to be detached to the work component. The *rstype2* parameter must be one of the following defined in rset.h.

R_RSET
Resource set attachment

R_SRADID
SRADID attachment

rsid2 Specifies the resource associated with the *rstype2* parameter. The *rsid2* parameter is ignored for R_RSET and R_SRADID *rstype2* resource types.

flags All flags bits are reserved for future use and must be specified as 0.

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate an error.

Error Codes

Item	Description
EINVAL	One of the following occurred: • The *flags* parameter contains an invalid value. • The *rstype1* or *rstype2* parameter contains an invalid type identifier.
ESRCH	A work component specified by the *rstype1* and *rsid1* parameters does not exist.
ENOTSUP	One of the following occurred: • An attempt to detach an SRADID (Scheduler Resource Allocation Domain Identifier) is made and ENHANCED_AFFINITY is disabled. • An attempt to detach an SRADID to a file is made. • An R_SUBRANGE request with su_rstype R_PROCMEM is made and the su_rsid.at_pid field is not RS_MYSELF.
EPERM	One of the following occurred: • *rstype2* specified R_RSET and calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege. • *rstype2* specified R_RSET and calling process has neither root authority nor the same effective user ID as the process identified by the *rstype1* and *rsid1* parameters.

ra_detachrset Subroutine

Purpose

Detaches a work component from a resource set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int ra_detachrset (rstype, rsid, flags)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
```

Description

The **ra_detachrset** subroutine detaches a work component specified by *rstype* and *rsid* from a resource set.

The work component is an existing process identified by the process ID or an existing kernel thread identified by the kernel thread ID (tid). A process ID or thread ID value of RS_MYSELF indicates the detach command applies to the current process or the current kernel thread, respectively.

The following conditions must be met to detach a process or a kernel thread from a resource set:

- The calling process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling process must either have root authority or the same effective userid as the target process.
- For R_FILDES *rstype*, the calling process must specify an open file descriptor, and it must have write access to the file, or the calling process' effective userid must be equal to the file owner's userid.
- For R_SHM *rstype*, the calling process' effective userid must be equal to the shared segment's owner.

If these conditions are not met, the operation will fail.

Once a process is detached from a resource set, the threads in the process can run on all available processors contained in the process' partition resource set. Once a kernel thread is detached from a resource set, that thread can run on all available processors contained in its process effective or partition resource set.

Parameters

Item	Description
rstype	Specifies the type of work component to be detached from to the resource set specified by *rset*. This parameter must be the following value, defined in **rset.h**: • R_PROCESS: existing process • R_THREAD: existing kernel thread • R_FILDES: file identified by an open file descriptor • R_SHM: shared memory segment identified by shared memory segment ID • R_SUBRANGE: attachment involves a subrange of the work component
rsid	Identifies the work component to be attached to the resource set specified by *rset*. This parameter must be the following: • Process ID (for *rstype* of R_PROCESS): set the *rsid_t at_pid* field to the desired process' process ID. • Kernel thread ID (for *rstype* of R_THREAD): set the *rsid_t at_tid* field to the desired kernel thread's thread ID. • Open file descriptor (for *rstype* of R_FILDES): set the *rsid_t at_fd* field to the desired file descriptor. • Shared memory segment ID (for *rstype* of R_SHM): set the *rsid_t at_shmid* field to the desired shared memory ID. • Pointer to a `subrange_t` struct (for *rstype* of R_SUBRANGE): set the `subrange_t` *su_offset*, *su_length*, *su_rstype*, and *su_rsid* fields. The other fields in the `subrange_t` struct are ignored.
flags	For *rstype* of R_PROCESS, the R_DETACH_ALLTHRDS indicates that R_THREAD *rsets* are detached from all threads in a specified process. The process' effective *rset* is not detached in this case. Reserved for future use. Specify as `0`.

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **ra_detachrset** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *flags* parameter contains an invalid value. • The *rstype* parameter contains an invalid type qualifier.
ESRCH	The process or kernel thread identified by the *rstype* and *rsid* parameters does not exist.
EPERM	One of the following is true: • The calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege. • The calling process has neither root authority nor the same effective user ID as the process identified by the *rstype* and *rsid* parameters.

ra_exec Subroutine

Purpose

Executes a file and attaches it to a given resource.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int ra_execl(rstype, rsid, flags, path, argument0 [,argument1,...], 0)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
const char * path, argument0, argument1,...;
```

```
int ra_execle(rstype, rsid, flags, path, argument0[,argument1,...], 0, envptr)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
const char * path, argument0, argument1,...;
char * const envptr[];
```

```
int ra_execlp(rstype, rsid, flags, File, argument0[,argument1,...], 0)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
const char * File, argument0, argument1,...;
```

```
int ra_execv (rstype, rsid, flags, path, argumentv)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
const char * path;
char * const argumentv[];
```

```
int ra_execve (rstype, rsid, flags, path, argumentv, envptr)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
const char * path;
char * const argumentv[], envptr[];
```

```
int ra_execvp (rstype, rsid, flags, File, argumentv)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
```

```
const char * File;
char * const argumentv[];
```

```
int ra_exect(rstype, rsid, flags, path, argumentv, envptr)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
char * path, argumentv, envptr[];
```

Description

The **ra_exec** subroutine in all its forms, executes a new program in the calling process, and attaches the process to the resource specified by the *rstype* and *rsid* parameters. The **ra_exec** subroutine can attach the new process to a resource set (rstype R_RSET) or to an sradid (rstype R_SRADID).

The following conditions must be met to successfully attach a process to a resource set:

- The resource set must contain processors that are available in the system.
- The process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling thread must not have a bindprocessor binding to a processor.
- The resource set must be contained in (be a subset of) the process' partition resource set.

Note: When the **exec** subroutine is used, the new process image inherits its process' resource set attachments.

Dynamic Processor Deallocation and DLPAR may invalidate the processor attachment that is being specified. A program must become DLPAR Aware to resolve this problem.

The *flags* parameter can be set to indicate the policy for using the resources contained in the resource set specified in the *rset* parameter. The only supported scheduling policy is R_ATTACH_STRSET, which is useful only when the processors of the system are running in simultaneous multithreading mode. Processors like the POWER5 support simultaneous multithreading, where each physical processor has two execution engines, called *hardware threads*. Each hardware thread is essentially equivalent to a single processor, and each is identified as a separate processor in a resource set. The R_ATTACH_STRSET flag indicates that the process is to be scheduled with a single-threaded policy; namely, that it should be scheduled on only one hardware thread per physical processor. If this flag is specified, then all of the available processors indicated in the resource set must be of exclusive use (the processor must belong to some exclusive use processor resource set). A new resource set, called an *ST resource set*, is constructed from the specified resource set and attached to the process according to the following rules:

- All offline processors are ignored.
- If all the hardware threads (processors) of a physical processor (when running in simultaneous multithreading mode, there will be more than one active hardware thread per physical processor) are not included in the specified resource set, the other processors of the processor are ignored when constructing the ST resource set.
- Only one processor (hardware thread) resource per physical processor is included in the ST resource set.

Parameters

The **ra_exec** subroutine has the same parameters as the **exec** subroutine, with the addition of the following new parameters:

Item	Description
rstype	Specifies the type of resource the new process image will be attached to. This parameter must be one of the following: • R_RSET: resource set • R_SRADID: sradid

Item	Description
rsid	Identifies the resource the new process image will be attached to: • Resource set handle (for *rstype* R_RSET): set the rsid.at_rset field to the desired resource set. • SRADID (Scheduler Resource Allocation Domain Identifier for rstype R_SRADID): set the rsid.at_sradid field to the desired sradid.
flags	Specifies the policy to use for the process. For rstype R_RSET, the R_ATTACH_STRSET flag indicates that the process is to be scheduled with a single-threaded policy (only on one hardware thread per physical processor). All other flag bits are reserved and must be specified as 0.

Return Values

The **ra_exec** subroutine's return values are the same as the **exec** subroutine's return values.

Error Codes

The **ra_exec** subroutine's error codes are the same as the **exec** subroutine's error codes, with the addition of the following error codes:

Item	Description
EINVAL	One of the following is true: • The *rstype* parameter contains an invalid type identifier. • The *flags* parameter contains an invalid flags value. • The R_ATTACH_STRSET *flags* parameter is specified and one or more processors in the *rset* parameter are not assigned for exclusive use.
ENODEV	The resource set specified by the *rset* parameter does not contain any available processors, or the R_ATTACH_STRSET *flags* parameter is specified and the constructed ST resource set does not have any available processors.
ENODEV	An invalid rsid SRADID is specified.
EFAULT	Invalid address.
EPERM	One of the following is true: • The calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege. • The calling process contains one or more threads with a bindprocessor processor binding. • The specified resource set is not included in the calling process' partition resource set.
ENOTSUP	An attempt to attach an SRADID is made and ENHANCED_AFFINITY is disabled.

ra_fork Subroutine

Purpose

Creates and attaches a new process to a given resource.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
pid_t ra_fork(rstype, rsid, flags)
```

```
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
```

Description

The **ra_fork** subroutine creates a new process, and attaches the new process to the resource specified by the *rstype* and *rsid* parameters. The **ra_fork** subroutine attaches the new process to a resource set (rstype R_RSET) or to an sradid (rstype R_SRADID).

The following conditions must be met to successfully attach a process to a resource set:

- The resource set must contain processors that are available in the system.
- The process must either have root authority or have CAP_NUMA_ATTACH capability.
- The calling thread must not have a bindprocessor binding to a processor.
- The resource set must be contained in (be a subset of) the process' partition resource set.

Note: When the **fork** subroutine is used, the child process inherits its parent's resource set attachments.

Dynamic Processor Deallocation and DLPAR may invalidate the processor attachment that is being specified. A program must become DLPAR Aware to resolve this problem.

The *flags* parameter can be set to indicate the policy for using the resources contained in the resource set specified in the *rset* parameter. The only supported scheduling policy is R_ATTACH_STRSET, which is useful only when the processors of the system are running in simultaneous multithreading mode. Processors like the POWER5 support simultaneous multithreading, where each physical processor has two execution engines, called *hardware threads*. Each hardware thread is essentially equivalent to a single processor, and each is identified as a separate processor in a resource set. The R_ATTACH_STRSET flag indicates that the process is to be scheduled with a single-threaded policy; namely, that it should be scheduled on only one hardware thread per physical processor. If this flag is specified, then all of the available processors indicated in the resource set must be of exclusive use (the processor must belong to some exclusive use processor resource set). A new resource set, called an *ST resource set*, is constructed from the specified resource set and attached to the process according to the following rules:

- All offline processors are ignored.
- If all the hardware threads (processors) of a physical processor (when running in simultaneous multithreading mode, there will be more than one active hardware thread per physical processor) are not included in the specified resource set, the other processors of the processor are ignored when constructing the ST resource set.
- Only one processor (hardware thread) resource per physical processor is included in the ST resource set.

Parameters

Item	Description
rstype	Specifies the type of resource the new process will be attached to. This parameter must be one the following: • R_RSET: resource set. • R_SRADID: sradid
rsid	Identifies the resource the new process will be attached to: • Resource set handle (for rstype R_RSET): sets the rsid.at_rset field to the desired resource set. • SRADID (Scheduler Resource Allocation Domain Identifier for rstype R_SRADID): sets the rsid.at_sradid field to the desired sradid.

Item	Description
flags	Specifies the policy to use for the process. For rstype R_RSET, the R_ATTACH_STRSET flag indicates that the process is to be scheduled with a single-threaded policy (only on one hardware thread per physical processor). All other flag bits are reserved and must be specified as 0.

Return Values

The **ra_fork** subroutine's return values are the same as the **fork** subroutine's return values.

Error Codes

The **ra_fork** subroutine's error codes are the same as the **fork** subroutine's error codes with the addition of the following:

Item	Description
EINVAL	One of the following is true: • The *rstype* parameter contains an invalid type identifier. • The *flags* parameter contains an invalid flags value. • The R_ATTACH_STRSET *flags* parameter is specified and one or more processors in the *rset* parameter are not assigned for exclusive use.
ENODEV	The resource set specified by the *rset* parameter does not contain any available processors, or the R_ATTACH_STRSET *flags* parameter is specified and the constructed ST resource set does not have any available processors.
ENODEV	An invalid *rsid* SRADID is specified.
EFAULT	Invalid address.
EPERM	One of the following is true: • The calling process has neither root authority nor CAP_NUMA_ATTACH attachment privilege. • The calling process contains one or more threads with a bindprocessor processor binding. • The specified resource set is not included in the calling process' partition resource set.
ENOTSUP	An attempt to attach an SRADID is made and ENHANCED_AFFINITY is disabled.

ra_free_attachinfo Subroutine

Purpose

Frees the memory allocated for the attachment information returned by `ra_get_attachinfo`.

Library

Standard C library (`libc.a`)

Syntax

```
#include <sys/rset.h>

int ra_free_attachinfo_t(info)
attachinfo_t *info;
```

Description

The `ra_free_attachinfo` subroutine frees the memory allocated by `ra_get_attachinfo` to contain the `attachinfo_t` structures returning the attachment information.

Parameters

Item	Description
info	Pointer to the `attachinfo_t` structure that was returned by a previous call to `ra_get_attachinfo`.

Return Values

On successful completion, a value of `0` is returned. Otherwise, a value of `-1` is returned and the `errno` global variable is set to indicate the error.

Error Codes

The `ra_free_attachinfo` subroutine is unsuccessful if the following is true:

Item	Description
`EINVAL`	The *info* parameter is a null pointer.

ra_get_attachinfo Subroutine

Purpose

Retrieves the resource set attachments to which a work component is attached.

Library

Standard C library (`libc.a`)

Syntax

```
#include <sys/rset.h>

attachinfo_t *ra_get_attachinfo(rstype, rsid, offset, length, flags)
rstype_t rstype;
rsid_t rsid;
off64_t offset;
size64_t length;
unsigned int flags;
```

Description

The `ra_get_attachinfo` subroutine retrieves information describing the attachments involving the work component specified by *rstype* and *rsid*.

This information is returned as a null-terminated linked list of `attachinfo_t` structures. The `attachinfo_t` structures are allocated in the caller's process heap. The `ra_free_attachinfo` subroutine is provided to free the list of `attachinfo_t` structures returned by `ra_get_attachinfo`.

The `ra_get_attachinfo` subroutine retrieves attachment information for the following work components:

- A shared memory object identified by a shared memory segment ID.
- A file identified by an open file descriptor.

- An address range in the current user process.
- An address range in one of the above work components identified by its *offset* in the object and its *length*.

If *rstype* is a memory object and *length* has a `0` value, the attachment information returned is for the last portion of the memory object, beginning with *offset*.

Note: Resource set attachments can change during or after `ra_get_attachinfo` retrieves them. There is no guarantee that the returned attachments still exist, or that all existing attachments were retrieved.

Parameters

Item	Description
rstype	Specifies the type of work component for which the attachment information is to be retrieved. This parameter can have one of the following values: **R_SHM** Attachment information of a shared memory, identified by its shared memory identifier, is to be retrieved. **R_FILDES** Attachment information of a file, identified by its open file descriptor, is to be retrieved. **R_PROCMEM** Attachment information of a memory range in the user process is to be retrieved.
rsid	Identifies the work component for which the attachment information is to be retrieved. This parameter can be one of the following: • shared memory segment ID (if the value of *rstype* is R_SHM) • open file descriptor (if the value of *rstype* is R_FILDES) • RS_MYSELF (if value of rstype is R_PROCMEM)
offset	Specifies the offset of a range within a memory object for which the attachment information is to be retrieved. This parameter is taken into account only for the following values of *rstype*: • R_SHM: starting offset within the shared memory object identified by *rsid* • R_FILDES: absolute offset within the file identified by *rsid* • R_PROCMEM: starting offset of memory range in user process.

Item	Description
length	Specifies the length of a range within a memory object for which the attachment information is to be retrieved. This parameter is taken into account only for the following values of rstype: • R_SHM: length of a range within the shared memory object identified by *rsid* • R_FILDES: length of a range within the file identified by *rsid* • R_PROCMEM: length of range in user process.
flags	Reserved for future use. Specify as `0`.

Return Values

On successful completion, a pointer to the first element in a null-terminated list of `attachinfo_t` structures is returned. A null pointer is returned if the work component does not have any attachments. Otherwise, a value of `-1` is returned and the `errno` global variable is set to indicate the error.

Error Codes

The `ra_get_attachinfo` subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following conditions is true: • The *flags* parameter contains an invalid value. • The *rstype* parameter contains an invalid type qualifier. • The *rstype* parameter is R_SHM and *rsid* is not a valid shared memory segment.
EBADF	The *rstype* parameter is R_FILDES and *rsid* is not a valid open file descriptor.
ENOTSUP	The *rstype* parameter is R_PROCMEM and rsid.at_pid field is not RS_MYSELF.

ra_getrset Subroutine

Purpose

Gets the resource set to which a work component is attached.

Library

Standard C library (**libc.a**)

Syntax

```
# include <sys/rset.h>
int ra_getrset (rstype, rsid, flags, rset)
rstype_t rstype;
rsid_t rsid;
unsigned int flags;
rsethandle_t rset;
```

Description

The **ra_getrset** subroutine returns the resource set to which a specified work component is attached.

The work component is an existing process identified by the process ID or an existing kernel thread identified by the kernel thread ID (tid). A process ID or thread ID value of RS_MYSELF indicates the resource set attached to the current process or the current kernel thread, respectively, is requested.

The following return values from the **ra_getrset** subroutine indicate the type of resource set returned:

- A value of RS_EFFECTIVE_RSET indicates the process was explicitly attached to the resource set. This may have been done with the **ra_attachrset** subroutine.
- A value of RS_PARTITION_RSET indicates the process was not explicitly attached to a resource set. However, the process had an explicitly set partition resource set. This may be set with the **rs_setpartition** subroutine or through the use of Workload Manager (WLM) work classes with resource sets.
- A value of RS_DEFAULT_RSET indicates the process was not explicitly attached to a resource set nor did it have an explicitly set partition resource set. The system default resource set is returned.
- A value of RS_THREAD_RSET indicates the kernel thread was explicitly attached to the resource set. This might have been done with the `ra_attachrset` subroutine.
- A value of RS_THREAD_PARTITION_RSET indicates that the kernel thread was not explicitly attached to a resource set. However, the thread had an explicitly set partition resource set. This was set through the use of WLM work classes with resource sets.

Parameters

Item	Description
rstype	Specifies the type of the work component whose resource set attachment is requested. This parameter must be the following value, defined in **rset.h**: • R_PROCESS: existing process • R_THREAD: existing kernel thread
rsid	Identifies the work component whose resource set attachment is requested. This parameter must be the following: • Process ID (for *rstype* of R_PROCESS): set the *rsid_t at_pid* field to the desired process' process ID. • Kernel thread ID (for *rstype* of R_THREAD): set the *rsid_t at_tid* field to the desired kernel thread's thread ID.
flags	Reserved for future use. Specify as 0.
rset	Specifies the resource set to receive the work component's resource set.

Return Values

If successful, a value of RS_EFFECTIVE_RSET, RS_PARTITION_RSET, RS_THREAD_RSET, RS_THREAD_PARTITION_RSET, or RS_DEFAULT_RSET is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **ra_getrset** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *flags* parameter contains an invalid value. • The *rstype* parameter contains an invalid type qualifier.
EFAULT	Invalid address.
ESRCH	The process or kernel thread identified by the *rstype* and *rsid* parameters does not exist.

ra_mmap or ra_mmapv Subroutine

Purpose

Maps a file or anonymous memory region into the process-address space and attaches the file or memory region to a given resource.

Library

Standard C Library (`libc.a`)

Syntax

```
#include <sys/rset.h>
#include <sys/mman.h>

void * ra_mmap( addr, len, prot, flags, fildes, off, rstype, rsid, policy )
void *addr;
off64_t len;
int prot;
int flags;
int fildes;
off64_t off;
rstype_t rstype;
rsid_t rsid;
unsigned int policy;

void * ra_mmapv( addr, len, prot, flags, fildes, off, rangecnt, rangevec )
void *addr;
off64_t len;
int prot;
int flags;
int fildes;
off64_t off;
int rangecnt;
subrange_t *rangevec;
```

Description

The `ra_mmap` subroutine maps the file or memory region, specified by *mmap_params*, into the process-address space and attaches it to the resource set specified by *rstype* and *rsid*. The resource set specified for attachment defines the resource allocation domains (RADs) from which the mapping's memory demands should be fulfilled. If the file or memory region is attached to a resource set specifying multiple RADs, its memory allocation is distributed among these RADs according to *policy*.

If a file is being mapped, the attachment for the new mapped region is reflected down to the portion of the file it maps and persists after the region is unmapped. The file's attachment persists until the last `close` of the file.

The `ra_mmapv` subroutine is similar to the `ra_mmap` subroutine, and allows multiple subranges of a file or memory region to be attached to different resource sets in a single `ra_mmapv` call.

The *rangecnt* argument specifies the number of subranges being mapped. The *rangevec* argument is a pointer to an array of `subrange_t` structures describing the attachments to be performed. Each

subrange_t structure specifies a portion of the file or memory region and the resource set to which the portion should be attached. If overlapping subranges are specified, ra_mmapv does not fail, but its behavior is undefined.

Child processes inherit all mapped regions and their resource set attachments from the parent process when the fork subroutine is called. The child process also inherits the same sharing and protection attributes for these mapped regions. A successful call to any exec subroutine unmaps all mapped regions created with the ra_mmap subroutine.

Attachments to a given RAD do not attach the process to the processors in that RAD. Attachments are only advisory; memory from a different RAD can be provided if the demand cannot be fulfilled from the RAD specified.

If overlapping subranges are mapped with attachments, the memory placement of the mapped regions is undefined.

The *su_rsoffset* and *su_rslength* fields of the subrange_t structures must be set to 0. Otherwise, ra_mmapv fails with EINVAL.

Parameters

Item	Description
addr	Specifies the starting address of the memory region to be mapped. When the MAP_FIXED flag is specified, this address must be a multiple of the page size returned by the sysconf subroutine using the _SC_PAGE_SIZE value for the *Name* parameter. A region is never placed at address 0, or at an address where it would overlap an existing region.
fildes	Specifies the file descriptor of the file-system object or of the shared memory object to be mapped. If the MAP_ANONYMOUS flag is set, the *fildes* parameter must be -1. After the successful completion of the ra_mmap or ra_mmapv subroutine, the file or the shared memory object specified by the *fildes* parameter can be closed without affecting the mapped region or the contents of the mapped file. Each mapped region creates a file reference, similar to an open file descriptor, which prevents the file data from being deallocated.
flags	Specifies attributes of the mapped region. Values for the *flags* parameter are constructed by a bitwise-inclusive ORing of values from the following list of symbolic names defined in the sys/mman.h file: **MAP_FILE** Specifies the creation of a new mapped file region by mapping the file associated with the *fildes* file descriptor. The mapped region can extend beyond the end of the file, both at the time when the ra_mmap subroutine is called and while the mapping persists. This situation could occur if a file with no contents was created just before the call to the ra_mmap subroutine, or if a file was later truncated. However, references to whole pages following the end of the file result in the delivery of a SIGBUS signal. Only one of the MAP_FILE and MAP_ANONYMOUS flags must be specified with the ra_mmap or ra_mmapv subroutine. **MAP_ANONYMOUS** Specifies the creation of a new, anonymous memory region that is initialized to all zeros. This memory region can be shared only with the descendants of the current process. When using this flag, the *fildes* parameter must be -1. Only one of the MAP_FILE and MAP_ANONYMOUS flags must be specified with the ra_mmap or ra_mmapvsubroutine. **MAP_VARIABLE** Specifies that the system select an address for the new memory region if the new memory region cannot be mapped at the address specified by the *addr* parameter, or if the *addr* parameter is null. Only one of the MAP_VARIABLE and MAP_FIXED flags must be specified with the ra_mmap or ra_mmapv subroutine. **MAP_FIXED** Specifies that the mapped region be placed exactly at the address specified by the *addr* parameter. If the application has requested SPEC1170 complaint behavior and the ra_mmap or ra_mmapv request is successful, the mapping replaces any previous mappings for the process' pages in the specified range. If the application has not requested SPEC1170 compliant behavior and a previous mapping exists in the range, the request fails. Only one of the MAP_VARIABLE and MAP_FIXED flags must be specified with the ra_mmap or ra_mmapv subroutine. **MAP_SHARED** When the MAP_SHARED flag is set, modifications to the mapped memory region will be visible to other processes that have mapped the same region using this flag. If the region is a mapped file region, modifications to the region will be written to the file. You can specify only one of the MAP_SHARED or MAP_PRIVATE flags with the ra_mmap or ra_mmapv subroutine. MAP_PRIVATE is the default setting when neither flag is specified unless you request SPEC1170 compliant behavior. In this case, you must choose either MAP_SHARED or MAP_PRIVATE. **MAP_PRIVATE** When the MAP_PRIVATE flag is specified, modifications to the mapped region by the calling process are not visible to other processes that have mapped the same region. If the region is a mapped file region, modifications to the region are not written to the file. If this flag is specified, the initial write reference to an object page creates a private copy of that page and redirects the mapping to the copy. Until then, modifications to the page by processes that have mapped the same region with the MAP_SHARED flag are visible. You can specify only one of the MAP_SHARED or MAP_PRIVATE flags with the ra_mmap or ra_mmapv subroutine. MAP_PRIVATE is the default setting when neither flag is specified unless you request SPEC1170 compliant behavior. In this case, you must choose either MAP_SHARED or MAP_PRIVATE.

Item	Description
len	Specifies the length, in bytes, of the memory region to be mapped. The system performs mapping operations over whole pages only. If the *len* parameter is not a multiple of the page size, the system will include in any mapping operation the address range between the end of the region and the end of the page containing the end of the region.
off	Specifies the file byte offset at which the mapping starts. This offset must be a multiple of the page size returned by the `sysconf` subroutine using the `_SC_PAGE_SIZE` value for the *Name* parameter.

Item	Description
policy	Specifies an advisory memory allocation policy that is to be applied. This parameter must have one of the following values defined in `sys/rset.h`: **P_FIRST_TOUCH** First Access memory policy. Memory is allocated from the RAD of the processor on which it is accessed the first time if this RAD is in the attachment resource set. Otherwise, memory is allocated from any RAD with memory available to the processor. **P_BALANCED** Balanced memory policy. Memory is allocated in a round robin manner across the RADs contained in the attachment resource set. **P_DEFAULT** Default memory placement policy.
prot	Specifies the access permissions for the mapped region. The `sys/mman.h` file defines the following access options: **PROT_READ** Region can be read. **PROT_WRITE** Region can be written. **PROT_EXEC** Region can be executed. **PROT_NONE** Region cannot be accessed. The *prot* parameter can be the `PROT_NONE` flag, or any combination of the `PROT_READ` flag, `PROT_WRITE` flag, and `PROT_EXEC` flag logically ORed together. If the `PROT_NONE` flag is not specified, access permissions can be granted to the region in addition to those explicitly requested. However, write access will not be granted unless the `PROT_WRITE` flag is specified. **Note:** The operating system generates a `SIGSEGV` signal if a program attempts an access that exceeds the access permission given to a memory region. For example, if the `PROT_WRITE` flag is not specified and a program attempts a write access, a `SIGSEGV` signal results. If the region is a mapped file that was mapped with the `MAP_SHARED` flag, the `ra_mmap` or `ra_mmapv` subroutine grants read or execute access permission only if the file descriptor used to map the file was opened for reading. It grants write access permission only if the file descriptor was opened for writing. If the region is a mapped file that was mapped with the `MAP_PRIVATE` flag, the `ra_mmap` or `ra_mmapv` subroutine grants read, write, or execute access permission only if the file descriptor used to map the file was opened for reading. If the region is an anonymous memory region, the `ra_mmap` or `ra_mmapv` subroutine grants all requested access permissions.
rangecnt	Specifies the number of `subrange_t` structures pointed to by *rangevec*.
rangevec	Specifies a pointer to an array of `subrange_t` structures describing the desired subrange attachments.

Item	Description
rsid	Identifies the resource to be attached to the file or memory region. All attachments are advisory. If memory cannot be allocated from the RADs identified by the resource, memory is allocated from any RAD in the system. • Resource set handle (for *rstype* **R_RSET**): set the rsid.at_rset field to the desired resource set. • SRADID (Scheduler Resource Allocation Domain Identifier for *rstype* **R_SRADID**): set the rsid.at_sradid field to the desired sradid.
rstype	Specifies the type of resource the file or memory region is to be attached to. This parameter must have one of the following values: • **R_RSET**: Resource set attachment • **R_SRADID**: SRADID attachment The **MAP_ANONYMOUS** *flags* field must be specified if *rstype* **R_SRADID** is specified.

Return Values

Upon successful completion, an address to the mapped file or memory region is returned. Otherwise, a value of `-1` is returned and the `errno` global variable is set to indicate the error.

Error Codes

Item	Description
EACCES	The file referred to by the *fildes* parameter is not open for read access, or the file is not open for write access and the **PROT_WRITE** flag was specified for a **MAP_SHARED** mapping operation. Or, the file to be mapped has enforced locking enabled and the file is currently locked.
EAGAIN	The *fildes* parameter refers to a device that has already been mapped.
EBADF	The *fildes* parameter is not a valid file descriptor, or the **MAP_ANONYMOUS** flag was set and the *fildes* parameter is not `-1`.
EFBIG	The mapping requested extends beyond the maximum file size associated with *fildes*.
EINVAL	The *flags* or *prot* parameter is invalid, or the *addr* parameter or *off* parameter is not a multiple of the page size returned by the **sysconf** subroutine using the **_SC_PAGE_SIZE** value for the *Name* parameter.
EINVAL	The application has requested SPEC1170 compliant behavior and the value of flags is invalid (neither **MAP_PRIVATE** nor **MAP_SHARED** is set).
EINVAL	The `subrange_t` structure specifies an invalid range.
EINVAL	The *su_rsoffset* and *su_rslength* fields of a `subrange_t` do not have a value of `0`.
EINVAL	The resource type is invalid (is not of type `R_RSET`).
EINVAL	The application has requested SPEC1170 compliant behavior and the value of *flags* is invalid (neither `MAP_PRIVATE` nor `MAP_SHARED` is set).
EMFILE	The application has requested SPEC1170 compliant behavior and the number of mapped regions would exceed an implementation-dependent limit (per process or per system).
ENODEV	The *fildes* parameter refers to an object that cannot be mapped, such as a terminal.
ENODEV	An invalid rsid SRADID is specified.

Item	Description
ENOMEM	There is not enough address space to map *len* bytes, or the application has not requested Single UNIX Specification, Version 2 compliant behavior and the **MAP_FIXED** flag was set and part of the address-space range (*addr*, *addr+len*) is already allocated.
ENOSYS	The `ra_mmap` subroutine is not supported on the system.
ENOSYS	The file specified is of a type that does not support physical attachments.
ENOTSUP	An attempt to map a memory region with an SRADID attachment is made and **ENHANCED_AFFINITY** is disabled.
ENOTSUP	An attempt to map a file with an SRADID attachment was made.
ENXIO	The addresses specified by the range (*off*, *off+len*) are invalid for the *fildes* parameter.
EOVERFLOW	The mapping requested extends beyond the offset maximum for the file description associated with *fildes*.
EPERM	The calling process does not have the necessary attachment privileges.

ra_shmget and ra_shmgetv Subroutines

Purpose

Gets a shared memory segment and attaches it to a resource.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
#include <sys/shm.h>

int ra_shmget(key, size, flags, rstype, rsid, att_flags)
key_t key;
size64_t size;
int flags;
rstype_t rstype;
rsid_t rsid;
unsigned int att_flags;
int ra_shmgetv(key, size, flags, rangecnt, rangevec)
key_t key;
size64_t size;
int flags;
int rangecnt;
subrange_t *rangevec;
```

Parameters

As per existing **shmget** usage, plus the following new parameters:

Item	Description
rstype	Specifies the type of resource the new shared memory segment is to be attached to. This parameter must have one of the following values: • R_RSET: Resource set attachment • R_SRADID: SRADID attachment

Item	Description
rsid	Identifies the resource to which the new shared memory segment is to be attached. All attachments are advisory. If memory cannot be allocated from the RAD(s) specified by rstype/rsid parameters, memory is allocated from any RAD in the system that has memory available. • Resource set handle (for rstype R_RSET): set the rsid.at field to the desired resource set. • SRADID (Scheduler Resource Allocation Domain Identifier for rstype R_SRADID): set the rsid.at_sradid to the desired sradid.
att_flags	Specifies an advisory memory allocation policy that is to be applied to the new shared memory segment. This parameter must have one of the following values defined in **sys/rset.h**: • **P_FIRST_TOUCH:** First Access memory policy. Memory is allocated from the current node, the RAD of the processor on which it is accessed for the first time, if this RAD is in the attachment resource set. If it is not, memory is allocated form an undefined RAD in the attachment resource set. • **P_BALANCED:** Balanced memory policy. Memory is allocated in a round robin manner across the RADs contained in the attachment resource set. • **P_DEFAULT:** Default memory placement policy.
rangecnt	Specifies the number of **subrange_t** structures pointed to by *rangevec*.
rangevec	Specifies a pointer to an array of **subrange_t** structures describing the desired subrange attachments.

Description

The **ra_shmget** subroutine returns the shared memory identifier associated with the specified *key*, *size* and *flags* parameters, attaching it to the resource set (**R_RSET**) specified by *rstype*, and *rsid*. The **ra_shmget** subroutine supports the sradid attachments. If the shared memory is attached to a set of physical resources involving multiple resource allocation domains (RADs), its memory allocation is distributed among these RADs according to *att_flags*. In an R_RSET type attachment, the processors specified in the input resource set are used for memory associativity; the resource set memory regions are ignored. All memory allocation attachments and policies are advisory.

If the new shared memory segment is to be attached in its entirety to a resource (that is, no subranges are involved), then the *rstype* or *rsid* parameters identify the memory attachment.

The **ra_shmgetv** subroutine is similar to the **ra_shmget** subroutine, and allows multiple subranges of the new shared memory segment to be attached to multiple resources in a single **ra_shmgetv** call. The *rangevec* argument is a pointer to an array of **subrange_t** structures describing the attachments to be performed. The *rangecnt* argument specifies the number of **subrange_t** structures pointed to by *rangevec*. All unused **subrange_t** structure fields, including those marked as reserved, must be initialized to the value of 0. Although it is not failing, the behavior with overlapping subranges is undefined.

Return Values

On successful completion, a shared memory identifier is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

As per existing **shmget** usage, plus the following errors:

Item	Description
EINVAL	One of the following conditions is true: • *rstype* contains an invalid type qualifier. • Invalid subrange fields. • *att_flags* contains an invalid flag.
EPERM	One of the following conditions is true: • The calling process has neither root authority nor CAP_NUMA_ATTACH privilege. • The resource specified by *rstype* and *rsid* is not included in the calling process's partition resource set.
ENODEV	An invalid *rsid* SRADID is specified.
ENOTSUP	An attempt to get a shared memory region with an SRADID attachment is made and ENHANCED_AFFINITY is disabled.

Examples

The following example attempts to use **ra_shmgetv** to create a **shmat** attachable shared memory region, whose first 32 megabytes are distributed using the P_BALANCED policy and the next 48 megabytes using the P_FIRST_TOUCH policy.

```
int flags, shm_id;
char *shm_at;
rsethandle_t rsetid;
subrange_t subranges[2] = { 0 };

rsetid = rs_alloc(RS_PARTITION);

subranges[0].su_offset = 0x0000000;
subranges[0].su_length = 0x2000000;
subranges[0].su_rstype = R_RSET;
subranges[0].su_rsid.at_rset = rsetid;
subranges[0].su_policy = P_BALANCED;

subranges[1].su_offset = 0x2000000;
subranges[1].su_length = 0x3000000;
subranges[1].su_rstype = R_RSET;
subranges[1].su_rsid.at_rset = rsetid;
subranges[1].su_policy = P_FIRST_TOUCH;

flags = (IPC_CREAT | SHM_PIN);
shm_id = ra_shmgetv (IPC_PRIVATE, 0x5000000, flags,
    sizeof(subranges) / sizeof(subrange_t), subranges
);
if (shm_id == -1)
{
    perror("ra_shmgetv failed!\n");
    exit(1);
}
```

Implementation Specifics

The **ra_shmget** and **ra_shmgetv** subroutines are part of the Base Operating System (BOS) Runtime.

ras_callback Registered Callback

Purpose

Component callback registered through the *ras_register* kernel service.

Syntax

```
kerrno_t (*ras_callback)(
              ras_block_t ras_blk,
              ras_cmd_t command,
              void *arg
              void *private_data);
```

Description

The component trace framework calls the **ras_callback** function each time an external event modifies a property of the component. Each component that calls the *ras_register* kernel service with a non-zero flags parameter must have the **ras_callback** registered callback function. Valid callback commands are those defined for individual RAS domains, such as Component Trace.

Note that the callback for a particular component does not have to be aware of, or act on, the children of the component as they have their own callbacks. Callbacks, in general, only do things relevant to the component for which they were called.

Parameters

Item	Description
ras_blk	The target control block pointer.
command	The command to act on. Commands are specific to a given RAS domain, such as Component Trace.
arg	Optional pointer to an argument needed for the given command.
private_data	Pointer to component-private data, specifically the pointer registered in the *ras_register* kernel service.

Return Values

ras_callback return 0 for success. Any other return value is a diagnostic error code from the component.

Execution Environment

Registrants must be aware that certain callbacks can be used at less than the interrupt priority of **INTBASE**, depending on what RAS domains the component is registered for. This depends on the designs for the domains involved. Because of the variability here, callbacks should be defined in a pinned object file.

rbac_chkauth Subroutine

Purpose

Perform a role-based access control (RBAC) authorization check.

Library

Security library (**libc.a**)

Syntax

```
#include <unistd.h>
int rbac_chkauth(username, authname, objname)
const char*username;
const char*authname;
const char*objnam;
```

Description

The **rbac_chkauth** function determines whether the specified username parameter has the authorization indicated by the authname parameter. The authname parameter represents a hierarchical naming structure in a string format for an authorization name. Only one authorization can be specified to describe the authorization hierarchy. If the username parameter is a null pointer or represents the same as a real user name of the calling process, and the specified authorization exists in the active role set of the process, the subroutine returns the value of 1. If the username parameter does not belong to the calling process, the subroutine checks the authorization in the user database. The objname parameter is not used in the subroutine.

You can use **rbac_chkauth** subroutine in the Enhanced (RBAC) mode only.

Parameters

username
Specifies the name of the user or a null pointer to use an real user ID of the calling process.

authname
Specifies the name of the authorization to be checked.

objname
Currently not used.

Return Values

The **rbac_chkauth** subroutine returns a 1 to indicate that the user has the specified authorization, or returns a 0 to indicate that the user does not have the specified authorization.

When the command fails, a value of -1 is returned and the errno value is set to indicate the error.

Error Codes

If the **rbac_chkauth** subroutine returns -1, one of the following errno values can be set:

Item	Description
EINVAL	The specified username parameter is invalid or authname parameter is a null pointer.
EPERM	The calling process does not have appropriate authority to verify the authname parameter for a user when the username parameter is a non-null pointer.

Example

The following example demonstrates how this subroutine is used:

```
#include <studio.h>
#include <errno.h>
#include <unistd.h>
#define SYSTEM_BOOT "aix.system.boot.reboot"

int boot_authcheck(void)
{
/*Verify whether this user (invoker) can perform system boot operation or not*/
switch (rbac_chkauth(NULL,SYSTEM_BOOT,NULL)) {
   case -1:
     perror("rbac_chkauth");
     return(0)
   case 0;
     fprint(stderr,"user is not authorized to perform system boot operation");
 }
 return(1);
}
```

read, readx, read64x, readv, readvx, eread, ereadv, pread, or preadv Subroutine

Purpose

Reads from a file.

Library

Item	Description
read, **readx**, **readv**, **readvx**, **read64x**, **pread**, **preadv**	Standard C Library (**libc.a**)
eread, **eread**	MLS library (**libmls.a**)

Syntax

```
#include <unistd.h>
```

```
ssize_t read (FileDescriptor, Buffer, NBytes)
int FileDescriptor;
void * Buffer;
size_t NBytes;
```

```
int readx (FileDescriptor, Buffer, NBytes, Extension)
int FileDescriptor;
char * Buffer;
unsigned int NBytes;
int Extension;
```

```
int read64x (FileDescriptor, Buffer, NBytes, Extension)
int FileDescriptor;
void *Buffer;
size_t NBytes;
void *Extension;
```

```
ssize_t pread (int fildes, void *buf, size_t nbyte, off_t offset);
```

```
#include <sys/uio.h>
```

```
ssize_t readv (FileDescriptor, iov, iovCount)
int FileDescriptor;
const struct iovec * iov;
int iovCount;
```

```
ssize_t readvx (FileDescriptor, iov, iovCount, Extension)
int FileDescriptor;
struct iovec *iov;
int iovCount;
int Extension;
```

```
#include <unistd.h>
#include <sys/uio.h>
```

```
ssize_t preadv (
int  FileDescriptor,
const struct iovec * iov,
int iovCount,
offset_t offset);
```

```
ssize_t eread (FileDescriptor, Buffer, Nbytes,  labels)
int FileDescriptor;
```

```
const void * Buffer;
size_t  NBytes;
sec_labels_t * labels;
```

```
ssize_t ereadv (FileDescriptor, iov, iovCount, labels)
int FileDescriptor;
const struct iovec * iov;
int iovCount;
sec_labels_t * labels;
```

Description

The **read** subroutine attempts to read *NBytes* of data from the file that is associated with the *FileDescriptor* parameter into the buffer pointed to by the *Buffer* parameter.

The **readv** subroutine performs the same action but scatters the input data into the *iovCount* buffers specified by the array of **iovec** structures pointed to by the *iov* parameter. Each **iovec** entry specifies the base address and length of an area in memory where data must be placed. The **readv** subroutine always fills an area completely before it proceeds to the next.

The **readx** and **readvx** subroutines are the same as the **read** and **readv** subroutines, respectively, with the addition of an *Extension* parameter, which is needed when reading from some device drivers and when reading directories. While directories can be read directly, the **opendir** and **readdir** calls be used instead, as it is a more portable interface.

On regular files and devices capable of seeking, the **read** starts at a position in the file that is given by the file pointer that is associated with the *FileDescriptor* parameter. Upon return from the **read** subroutine, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer that is associated with such a file is undefined.

On directories, the **readvx** subroutine starts at the position that is specified by the file pointer that is associated with the*FileDescriptor* parameter. The value of this file pointer must be either 0 or a value that the file pointer had immediately after a previous call to the **readvx** subroutine on this directory. Upon return from the **readvx** subroutine, the file pointer increments by a number that does not correspond to the number of bytes copied into the buffers.

When the system is attempting to read from an empty pipe (first-in-first-out (FIFO)):

- If no process has the pipe open for writing, the **read** returns 0 to indicate end-of-file.
- If some process, has the pipe open for writing:
  - If **O_NDELAY** and **O_NONBLOCK** are clear (the default), the **read** blocks until some data is written or the pipe is closed by all processes that open the pipe for writing.
  - If **O_NDELAY** is set, the **read** subroutine returns a value of 0.
  - If **O_NONBLOCK** is set, the **read** subroutine returns a value of **-1** and sets the global variable **errno** to **EAGAIN**.

When the system is attempting to read from a character special file that supports nonblocking reads, such as a terminal, and no data is available:

- If **O_NDELAY** and **O_NONBLOCK** are clear (the default), the **read** subroutine blocks until data becomes available.
- If **O_NDELAY** is set, the **read** subroutine returns 0.
- If **O_NONBLOCK** is set, the **read** subroutine returns **-1** and sets the **errno** global variable to **EAGAIN** if no data is available.

When the system is attempting to read a regular file that supports enforcement mode record locks, and all or part of the region to be read is locked by another process:

- If **O_NDELAY** and **O_NONBLOCK** are clear, the **read** blocks the calling process until the lock is released.
- If **O_NDELAY** or **O_NONBLOCK** is set, the **read** returns **-1** and sets the global variable **errno** to**EAGAIN**.

The behavior of an interrupted **read** subroutine depends on how the handler for the arriving signal was installed.

If the handler was installed, with an indication that subroutines must not be restarted, the **read** subroutine returns a value of **-1** and the global variable **errno** is set to **EINTR** (even if some data was already removed).

If the handler was installed, with an indication that subroutines must be restarted:

- If no data was read when the interrupt was handled, this **read** returns no value (it is restarted).
- If data was read when the interrupt was handled, this **read** subroutine returns the amount of data removed.

The **read64x** subroutine is the same as the **readx** subroutine, where the *Extension* parameter is a pointer to a **j2_ext** structure (see the **j2/j2_cntl.h** file). The **read64x** subroutine is used to read an encrypted file in raw mode (see O_RAW in the **fcntl.h** file). Using the O_RAW flag on encrypted files has the same limitations as using O_DIRECT on regular files.

The **eread** and **ereadv** subroutines read from the stream and retrieve the message. The **eread** subroutine copies the number of bytes of the data from the buffer to a stream associated with the *FileDescriptor* parameter. The *Nbyte* parameter specifies the number of bytes. The *Buffer* parameter points to the buffer. Security information is returned in the structure pointed to by the *labels* parameter.

The **pread** function performs the same action as **read**, except that it reads from a given position in the file without changing the file pointer. The first three arguments to **pread** are the same as **read** with the addition of a fourth argument that is offset for the wanted position inside the file. An attempt to perform a **pread** on a file that is incapable of seeking results in an error.

```
ssize_t pread64(int fildes , void *buf , size_t nbytes , off64_t offset)
```

The **pread64** subroutine performs the same action as **pread** but the limit of offset to the maximum file size for the file that is associated with the file Descriptor and DEV_OFF_MAX if the file associated with file Descriptor is a block special or character special file. If *fildes* refers to a socket, **read** is equivalent to the **recv** subroutine with no flags set.

Using the **read** or **pread** subroutine with a file descriptor obtained from a call to the **shm_open** subroutine fails with **ENXIO**.

The **preadv** subroutine performs the same action as the **readv** subroutine, except that the **preadv** subroutine reads from a given position in the file without changing the file pointer. The first three arguments of the **preadv** subroutine are the same as the **readv** subroutine with the addition of the *offset* argument that points to the position that you want inside the file. An error occurs when the file that the **preadv** subroutine reads from is incapable of seeking.

Parameters

Item	Description
FileDescriptor	A file descriptor that is identifying the object to be read.

Item	Description
Extension	Provides communication with character device drivers that require more information or return extra status. Each driver interprets the *Extension* parameter in a device-dependent way, either as a value or as a pointer to a communication area. Drivers must apply reasonable defaults when the value of the *Extension* parameter is 0. For directories, the *Extension* parameter determines the format in which directory entries must be returned: • If the value of the *Extension* parameter is 0, the format in which directory entries are returned depends on the value of the **real directory read** flag (described in the **ulimit** subroutine). • If the calling process does not have the **real directory read** flag set, the buffers are filled with an array of directory entries that are truncated to fit the format of the System V directory structure. This process provides compatibility with programs written for UNIX System V. • If the calling process has the **real directory read** flag set (see the **ulimit** subroutine), the buffers are filled with an image of the underlying implementation of the directory. • If the value of the *Extension* parameter is 1, the buffers are filled with consecutive directory entries in the format of a**dirent** structure. This process is logically equivalent to the **readdir** subroutine. • Other values of the *Extension* parameter are reserved. For tape devices, the *Extension* parameter determines the response of the **readx** subroutine when the tape drive is in variable block mode and the read request is for less than the tape's block size. • If the value of the *Extension* parameter is **TAPE_SHORT_READ**, the **readx** subroutine returns the number of bytes requested and sets the **errno** global variable to a value of 0. • If the value of the *Extension* parameter is 0, the **readx** subroutine returns a value of 0 and sets the **errno** global variable to **ENOMEM**.

iov	Points to an array of **iovec** structures that identifies the buffers into which the data is to be placed. The **iovec** structure is defined in the **sys/uio.h** file and contains the following members:
	
	`caddr_t iov_base;`
	`size_t iov_len;`

iovCount	Specifies the number of **iovec** structures pointed to by the *iov* parameter.
Buffer	Points to the buffer.
NBytes	Specifies the number of bytes read from the file that is associated with the*FileDescriptor* parameter. **Note:** When reading tapes, the **read** subroutines use a physical tape block on each call to the subroutine. If the physical data block size is larger than specified by the *Nbytes* parameter, an error is returned, since all of the data from the read does not fit into the buffer that is specified by the read. To avoid read errors that are caused by unknown blocking sizes on tapes, set the *NBytes* parameter to a large value (such as 32K bytes).
offset	The position in the file where the reading begins.
labels	Points to the extended security attribute structure.

Return Values

Upon successful completion, the **read**, **readx**, **read64x**, **readv**, **readvx**, **pread**, and **preadv** subroutines return the number of bytes read and placed into buffers. The system guarantees to read the number of bytes requested if the descriptor references a normal file that has the same number of bytes left before the end of the file is reached, but in no other case.

A value of 0 is returned when the end of the file is reached. (For information about communication files, see the **ioctl** and **termio** files.)

Otherwise, a value of **-1** is returned, the global variable **errno** is set to identify the error, and the content of the buffer pointed to by the *Buffer* or *iov* parameter is indeterminate.

Upon successful completion, the **eread** and **ereadv** subroutines return a value of 0. Otherwise, the global variable **errno** is set to identify the error.

Error Codes

The **read**, **readx**, **read64x**, **readv**, **readvx**, **pread**, **eread**, **ereadv**, and **preadv** subroutines are unsuccessful if one or more of the following are true:

Item	Description
EBADMSG	The file is a STREAM file that is set to control-normal mode and the message that is waiting to be read includes a control part.
EBADF	The *FileDescriptor* parameter is not a valid file descriptor open for reading.
EINVAL	The file position pointer that is associated with the *FileDescriptor* parameter was negative.
EINVAL	The sum of the **iov_len** values in the *iov* array was negative or overflowed a 32-bit integer.
EINVAL	The value of the *iovCount* parameter was not 1 - 16, inclusive.
EINVAL	The value of the *Nbytes* parameter that is larger than OFF_MAX, was requested on the 32-bit kernel. This issue is a case where the system call is requested from a 64-bit application that is running on a 32-bit kernel.

Item	Description
EINVAL	The STREAM or multiplexer that is referenced by *FileDescriptor* is linked (directly or indirectly) downstream from a multiplexer.
EAGAIN	The file was marked for non-blocking I/O, and no data was ready to be read.
EFAULT	The *Buffer* or part of the *iov* points to a location outside of the allocated address space of the process.
EFAULT	The user does not have authority to access the *Buffer*.
EDEADLK	A deadlock would occur if the calling process were to sleep until the region to be read was unlocked.
EINTR	A **read** was interrupted by a signal before any data arrived, and the signal handler was installed with an indication that subroutines are not to be restarted.
EIO	An I/O error occurred while reading from the file system.
EIO	The process is a member of a background process that is attempting to read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group has no parent process.
EFBIG	An offset greater than **MAX_FILESIZE** was requested on the 32-bit kernel.
ENXIO	The **read** or **pread** subroutine was used with a file descriptor obtained from a call to the **shm_open** subroutine.

Item	Description
EOVERFLOW	An attempt was made to read from a regular file where NBytes was greater than zero and the starting offset was before the end-of-file and was greater than or equal to the offset maximum established in the open file description that is associated with *FileDescriptor*.

The **read**, **readx**, **readv**, **readvx**, **pread**, and **preadv** subroutines might be unsuccessful if the following is true:

Item	Description
ENXIO	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
ESPIPE	*fildes* is associated with a pipe or FIFO.

If Network File System (NFS) is installed on the system, the **read** system call can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection that is timed out.

The **read64x** subroutine was unsuccessful if the **EINVAL** error code is returned:

Item	Description
EINVAL	The **j2_ext** structure was not initialized correctly. For example, the version was wrong, or the file was not encrypted.
EINVAL	The **j2_ext** structure was passed issuing the **J2EXTCMD_RDRAW** command for files that were not opened in raw-mode.

The **eread** and **ereadv** subroutines were unsuccessful if one of the following error codes is true:

Item	Description
ENOMEM	The memory or space is too small.
EACCES	Permission Denied. The user has insufficient privileges to read data.
ERESTART	**ERESTART** is used to determine if whether a system call is restartable or not.

The **readv** subroutine was unsuccessful if the following error code is true:

Item	Description
EINVAL	The value of the *iovCount* parameter is greater than 15.

readdir_r Subroutine

Purpose

Reads a directory.

Library

Thread-Safe C Library (**libc_r.a**)

Syntax

```
#include <sys/types.h>
#include <dirent.h>
```

int readdir_r (*DirectoryPointer***, *Entry*, *Result*)**
DIR * *DirectoryPointer*;
struct dirent * *Entry*;
struct dirent ** *Result*;

Description

The **readdir_r** subroutine returns the directory entry in the structure pointed to by the *Result* parameter. The **readdir_r** subroutine returns entries for the . (dot) and .. (dot-dot) directories, if present, but never returns an invalid entry (with `d_ino` set to `0`). When it reaches the end of the directory, the `readdir_r` subroutine returns 9 and sets the *Result* parameter to NULL. When it detects an invalid **seekdir** operation, the **readdir_r** subroutine returns a 9.

Note: The **readdir** subroutine is reentrant when an application program uses different *DirectoryPointer* parameter values (returned from the **opendir** subroutine). Use the **readdir_r** subroutine when multiple threads use the same directory pointer.

Using the **readdir_r** subroutine after the **closedir** subroutine, for the structure pointed to by the *DirectoryPointer* parameter, has an undefined result. The structure pointed to by the *DirectoryPointer* parameter becomes invalid for all threads, including the caller.

Programs using this subroutine must link to the **libpthreads.a** library.

Parameters

Item	Description
DirectoryPointer	Points to the **DIR** structure of an open directory.
Entry	Points to a structure that contains the next directory entry.
Result	Points to the directory entry specified by the *Entry* parameter.

Return Values

Item	Description
0	Indicates that the subroutine was successful.
9	Indicates that the subroutine was not successful or that the end of the directory was reached. If the user has set the environment variable `XPG_SUS_ENV=ON` prior to execution of the process, then the `SIGXFSZ` signal is posted to the process when exceeding the process' file size limit, and the subroutine will always be successful.

Error Codes

If the **readdir_r** subroutine is unsuccessful, the **errno** global variable is set to one of the following values:

Item	Description
EACCES	Search permission is denied for any component of the structure pointed to by the *DirectoryPointer* parameter, or read permission is denied for the structure pointed to by the *DirectoryPointer* parameter.

Item	Description
ENAMETOOLONG	The length of the *DirectoryPointer* parameter exceeds the value of the **PATH_MAX** variable, or a path-name component is longer than the value of **NAME_MAX** variable while the **_POSIX_NO_TRUNC** variable is in effect.
ENOENT	The named directory does not exist.
ENOTDIR	A component of the structure pointed to by the *DirectoryPointer* parameter is not a directory.
EMFILE	Too many file descriptors are currently open for the process.
ENFILE	Too many file descriptors are currently open in the system.
EBADF	The structure pointed to by the *DirectoryPointer* parameter does not refer to an open directory stream.

Examples

To search a directory for the entry name , enter:

```
len = strlen(name);
DirectoryPointer = opendir(".");
for (readdir_r(DirectoryPointer, &Entry, &Result); Result != NULL;
 readdir_r(DirectoryPointer, &Entry, &Result))
        if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {
                closedir(DirectoryPointer);
                return FOUND;
        }
closedir(DirectoryPointer);
return NOT_FOUND;
```

readlink or readlinkat Subroutine

Purpose

Reads the contents of a symbolic link.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
int readlink ( Path,  Buffer,  BufferSize)
const char *Path;
char *Buffer;
size_t BufferSize;

int readlinkat ( DirFileDescriptor, Path, Buffer, BufferSize )
int DirFileDescriptor;
const char * Path;
char * Buffer;
size_t BufferSize;
```

Description

The **readlink** and **readlinkat** subroutines copy the contents of the symbolic link named by the *Path* parameter in the buffer specified in the *Buffer* parameter. The *BufferSize* parameter indicates the size of the buffer in bytes. If the actual length of the symbolic link is less than the number of bytes specified in the *BufferSize* parameter, the string copied into the buffer will be null-terminated. If the actual length of the symbolic link is greater than the number of bytes specified in the *Buffersize* parameter, an error is returned. The length of a symbolic link cannot exceed 1023 characters or the value of the **PATH_MAX** constant. **PATH_MAX** is defined in the **limits.h** file.

The **readlinkat** subroutine is equivalent to the **readlink** subroutine if the *DirFileDescriptor* parameter is **AT_FDCWD** or *Path* is an absolute path name. If *DirFileDescriptor* is a valid file descriptor of an open directory and *Path* is a relative path name, *Path* is considered to be relative to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory.

If *DirFileDescriptor* was opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted for that directory by using the current permissions of the directory. If the directory was opened with the **O_SEARCH** open flag, the subroutine does not perform the check for that directory.

Parameters

Item	Description
DirFileDescriptor	Specifies the file descriptor of an open directory.
Path	Specifies the path name of the destination file or directory. If *DirFileDescriptor* is specified and *Path* is a relative path name, then *Path* is considered relative to the directory specified by *DirFileDescriptor*.
Buffer	Points to the user buffer. The buffer should be at least as large as the *BufferSize* parameter.
BufferSize	Indicates the size of the buffer. The contents of the link are null-terminated, provided there is room in the buffer.

Return Values

Upon successful completion, the **readlink** and **readlinkat** subroutines return a count of the number of characters placed in the buffer (not including any terminating null character). If the **readlink** or **readlinkat** subroutine is unsuccessful, the buffer is not modified, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **readlink** and **readlinkat** subroutines fail if one or both of the following are true:

Item	Description
ENOENT	The file named by the *Path* parameter does not exist, or the path points to an empty string.
EINVAL	The file named by the *Path* parameter is not a symbolic link.
ERANGE	The path name in the symbolic link is longer than the *BufferSize* value.

The **readlinkat** subroutine is unsuccessful if one or more of the following is true:

Item	Description
EBADF	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** not a valid file descriptor.

Item	Description
ENOTDIR	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.

The **readlink** and **readlinkat** subroutines can also fail due to additional errors.

If Network File System (NFS) is installed on the system, the **readlink** and **readlinkat** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

read_real_time, read_wall_time,time_base_to_time or mread_real time Subroutine

Purpose

Read the processor real-time clock or time base registers to obtain high-resolution elapsed time.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>
#include <sys/systemcfg.h>
```

```
int read_real_time(timebasestruct_t *t,
                   size_t size_of_timebasestruct_t);
```

```
int read_wall_time(timebasestruct_t *t,
                   size_t size_of_timebasestruct_t);
```

```
int time_base_to_time(timebasestruct_t *t,
                   size_t size_of_timebasestruct_t);
```

```
int mread_real_time(timebasestruct_t *t,
                   size_t size_of_timebasestruct_t);
```

Description

These subroutines are used for making high-resolution measurement of elapsed time, by using the processor real-time clock or time base registers. The **read_real_time** subroutine reads the value of the appropriate registers and stores them in a structure. The `read_wall_time` subroutine returns the monotonically increasing time base value. The **time_base_to_time** subroutine converts time base data to real time, if necessary. This process is divided into two steps because the process of reading the time is usually part of the timed code. The conversion from time base to real time can be moved out of the timed code.

The **read_real_time** subroutine reads the time base register. The *t* argument is a pointer to a *timebasestruct_t*, where the time values are recorded.

After the system calls the **read_real_time** subroutine, if it is running on a processor with a real-time clock, *t->tb_high* and *t->tb_low* contain the current clock values (seconds and nanoseconds), and *t->flag* contains the **RTC_POWER**.

If it is running on a processor with a time base register, *t->tb_high* and *t-tb_low* contain the current values of the time base register, and *t->flag* contains **RTC_POWER_PC**.

Note: The **read_real_time** subroutine occasionally provides negative timing results for MPI calls. Use the **mread_real_time** subroutine to monotonically increase timing values.

The **time_base_to_time** subroutine converts time base information to real time, if necessary. It is suggested that applications unconditionally call the **time_base_to_time** subroutine rather than conducting a check to see whether it is necessary.

If *t->flag* is **RTC_POWER**, the subroutine returns (the data is already in real-time format).

If *t->flag* is **RTC_POWER_PC**, the time base information in *t->tb_high* and *t->tb_low* is converted to seconds and nanoseconds; *t->tb_high* is replaced by the seconds; *t->tb_low* is replaced by the nanoseconds; and *t->flag* is changed to **RTC_POWER**.

Parameters

Item	Description
t	Points to a *timebasestruct_t*.

Return Values

The **read_real_time** subroutine returns **RTC_POWER** if the contents of the real-time clock are recorded in the *timebasestruct*, or returns **RTC_POWER_PC** if the content of the time base registers is recorded in the *timebasestruct*.

The `read_wall_time` subroutine always returns `RTC_POWER_PC`.

The **time_base_to_time** subroutine returns **0** if the conversion to real time is successful (or not necessary), otherwise **-1** is returned.

Examples

This example shows the time that it takes for **print_f** to print the comment between the begin and end time codes:

```
#include <stdio.h>
#include <sys/time.h>

int
main(void)
{
   timebasestruct_t start, finish;
   int val = 3;
   int secs, n_secs;

   /* get the time before the operation begins */
   read_real_time(&start, TIMEBASE_SZ);

   /* begin code to be timed */
   (void) printf("This is a sample line %d \n", val);
   /* end code to be timed   */

   /* get the time after the operation is complete */
   read_real_time(&finish, TIMEBASE_SZ);

   /*
    * Call the conversion routines unconditionally, to ensure
    * that both values are in seconds and nanoseconds regardless
    * of the hardware platform.
    */
   time_base_to_time(&start, TIMEBASE_SZ);
   time_base_to_time(&finish, TIMEBASE_SZ);

   /* subtract the starting time from the ending time */
   secs = finish.tb_high - start.tb_high;
   n_secs = finish.tb_low - start.tb_low;
```

```
  /*
   * If there was a carry from low-order to high-order during
   * the measurement, we may have to undo it.
   */
   if (n_secs < 0)  {
      secs--;
      n_secs += 1000000000;
      }

   (void) printf("Sample time was %d seconds %d nanoseconds\n",
                secs, n_secs);

   exit(0);
}
```

realpath Subroutine

Purpose

Resolves path names.

Library

Standard C Library **(libc.a)**

Syntax

#include <stdlib.h>

char *realpath (const char **file_name*, **char** **resolved_name*)

Description

The **realpath** subroutine performs filename expansion and path name resolution in *file_name* and stores it in *resolved_name.*

The **realpath** subroutine can handle both relative and absolute path names. For both absolute and relative path names, the **realpath** subroutine returns the resolved absolute path name.

The character pointed to by *resolved_name* must be big enough to contain the fully resolved path name. The value of PATH_MAX (defined in **limits.h** header file may be used as an appropriate array size.

Return Values

On successful completion, the **realpath** subroutine returns a pointer to the resolved name. Otherwise, it returns a null pointer, and sets **errno** to indicate the error. If the **realpath** subroutine encounters an error, the contents of *resolved_name* are undefined.

Error Codes

Under the following conditions, the **realpath** subroutine fails and sets **errno** to:

Item	Description
EACCES	Read or search permission was denied for a component of the path name.
EINVAL	*File_name* or *resolved_name* is a null pointer.
ELOOP	Too many symbolic links are encountered in translating *file_name*.

Item	Description
ENAMETOOLONG	The length of *file_name* or *resolved_name* exceeds PATH_MAX or a path name component is longer than NAME_MAX.
ENOENT	The *file_name* parameter does not exist or points to an empty string.
ENOTDIR	A component of the *file_name* prefix is not a directory.

The **realpath** subroutine may fail if:

Item	Description
ENOMEM	Insufficient storage space is available.

reboot Subroutine

Purpose

Restarts the system.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/reboot.h>

void reboot ( HowTo,  Argument)
int HowTo;
void *Argument;
```

Description

The **reboot** subroutine restarts or re-initial program loads (IPL) the system. The startup is automatic and brings up **/unix** in the normal, nonmaintenance mode.

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

The calling process must have root user authority in order to run this subroutine successfully.

Attention: Users of the **reboot** subroutine are not portable. The **reboot** subroutine is intended for use only by the **halt**, **reboot**, and **shutdown** commands.

Parameters

Item	Description
HowTo	Specifies one of the following values: **RB_SOFTIPL** Soft IPL. **RB_HALT** Halt operator; turn the power off. **RB_POWIPL** Halt operator; turn the power off. Wait a specified length of time, and then turn the power on.

Item	Description
Argument	Specifies the amount of time (in seconds) to wait between turning the power off and turning the power on. This option is not supported on all models. Please consult your hardware technical reference for more details.

Return Values

Upon successful completion, the **reboot** subroutine does not return a value. If the **reboot** subroutine fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **reboot** subroutine is unsuccessful if any of the following is true:

Item	Description
EPERM	The calling process does not have root user authority.
EINVAL	The *HowTo* value is not valid.
EFAULT	The *Argument* value is not a valid address.

re_comp or re_exec Subroutine

Purpose

Regular expression handler.

Library

Standard C Library (**libc.a**)

Syntax

```
char *re_comp( String)
const char *String;
```

```
int re_exec(String)
const char *String;
```

Description

Attention: Do not use the **re_comp** or **re_exec** subroutine in a multithreaded environment.

The **re_comp** subroutine compiles a string into an internal form suitable for pattern matching. The **re_exec** subroutine checks the argument string against the last string passed to the **re_comp** subroutine.

The **re_comp** subroutine returns 0 if the string pointed to by the *String* parameter was compiled successfully; otherwise a string containing an error message is returned. If the **re_comp** subroutine is passed 0 or a null string, it returns without changing the currently compiled regular expression.

The **re_exec** subroutine returns 1 if the string pointed to by the *String* parameter matches the last compiled regular expression, 0 if the string pointed to by the *String* parameter failed to match the last compiled regular expression, and -1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both **re_comp** and **re_exec** subroutines may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for the **ed** command, given the above difference.

Parameters

Item	Description
String	Points to a string that is to be matched or compiled.

Return Values

If an error occurs, the **re_exec** subroutine returns a -1, while the **re_comp** subroutine returns one of the following strings:

- No previous regular expression
- Regular expression too long
- unmatched \(
- missing]
- too many \(\) pairs
- unmatched \)

refresh or wrefresh Subroutine

Purpose

Updates the terminal's display and the curscr to reflect changes made to a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
refresh( )
```

wrefresh(*Window***)**
WINDOW ****Window*;**

Description

The **refresh** or **wrefresh** subroutines update the terminal and the curscr to reflect changes made to a window. The **refresh** subroutine updates the stdscr. The **wrefresh** subroutine refreshes a user-defined window.

Other subroutines manipulate windows but do not update the terminal's physical display to reflect their changes. Use the **refresh** or **wrefresh** subroutines to update a terminal's display after internal window representations change. Both subroutines check for possible scroll errors at display time.

Note: The physical terminal cursor remains at the location of the window's cursor during a refresh, unless the **leaveok** (“leaveok Subroutine” on page 847) subroutine is enabled.

The **refresh** and **wrefresh** subroutines call two other subroutines to perform the refresh operation. First, the **wnoutrefresh** (“doupdate, refresh, wnoutrefresh, or wrefresh Subroutines” on page 255) subroutine copies the designated window structure to the terminal. Then, the **doupdate** (“doupdate, refresh, wnoutrefresh, or wrefresh Subroutines” on page 255) subroutine updates the terminal's display and the cursor.

Parameters

Item	Description
Window	Specifies the window to refresh.

Examples

1. To update the terminal's display and the current screen structure to reflect changes made to the standard screen structure, use:

   ```
   refresh();
   ```

2. To update the terminal and the current screen structure to reflect changes made to a user-defined window called `my_window`, use:

   ```
   WINDOW *my_window;
   wrefresh(my_window);
   ```

3. To restore the terminal to its state at the last refresh, use:

   ```
   wrefresh(curscr);
   ```

   This subroutine is useful if the terminal becomes garbled for any reason.

regcmp or regex Subroutine

Purpose

Compiles and matches regular-expression patterns.

Libraries

Standard C Library (**libc.a**)

Programmers Workbench Library (**libPW.a**)

Syntax

```
#include <libgen.h>
```

```
char *regcmp ( String [, String, . . . ], (char *) 0)
const char *String, . . . ;

const char *regex ( Pattern,  Subject [,  ret, . . . ])
char *Pattern, *Subject, *ret, . . . ;
extern char *__loc1;
```

Description

Note: The `regcmp` and `regex` subroutines are provided for compatibility with existing applications only. For portable applications, use the `regcomp` and `regexec` subroutines instead.

The **regcmp** subroutine compiles a regular expression (or *Pattern*) and returns a pointer to the compiled form. The **regcmp** subroutine allows multiple *String* parameters. If more than one *String* parameter is given, then the **regcmp** subroutine treats them as if they were concatenated together. It returns a null pointer if it encounters an incorrect parameter.

You can use the **regcmp** command to compile regular expressions into your C program, frequently eliminating the need to call the **regcmp** subroutine at run time.

The **regex** subroutine compares a compiled *Pattern* to the *Subject* string. Additional parameters are used to receive values. Upon successful completion, the **regex** subroutine returns a pointer to the next unmatched character. If the **regex** subroutine fails, a null pointer is returned. A global character pointer, **__loc1**, points to where the match began.

The **regcmp** and **regex** subroutines are borrowed from the **ed** command; however, the syntax and semantics have been changed slightly. You can use the following symbols with the **regcmp** and **regex** subroutines:

Item	Description
[ ] * . ^	These symbols have the same meaning as they do in the **ed** command.
-	The minus sign (or hyphen) within brackets used with the **regex** subroutine means "through," according to the current collating sequence. For example, [a-z] can be equivalent to [abcd . . . xyz] or [aBbCc . . . xYyZz]. You can use the - by itself if the - is the last or first character. For example, the character class expression [ ] -] matches the] (right bracket) and - (minus) characters. The **regcmp** subroutine does not use the current collating sequence, and the minus sign in brackets controls only a direct ASCII sequence. For example, [a-z] always means [abc . . . xyz] and [A-Z] always means [ABC . . . XYZ] . If you need to control the specific characters in a range using the **regcmp** subroutine, you must list them explicitly rather than using the minus sign in the character class expression.
$	Matches the end of the string. Use the \n character to match a new-line character.
+	A regular expression followed by + (plus sign) means one or more times. For example, [0-9] + is equivalent to [0-9] [0-9] *.
{ *m*} {*m*,} {*m*, *u*}	Integer values enclosed in {} (braces) indicate the number of times to apply the preceding regular expression. The *m* character is the minimum number and the *u* character is the maximum number. The *u* character must be less than 256. If you specify only *m*, it indicates the exact number of times to apply the regular expression. {*m*,} is equivalent to {*m*,*u*} and matches *m* or more occurrences of the expression. The + (plus sign) and * (asterisk) operations are equivalent to {1,} and {0,}, respectively.

Item	Description
(...)$*n*	This stores the value matched by the enclosed regular expression in the (*n*+1)th *ret* parameter. Ten enclosed regular expressions are allowed. The **regex** subroutine makes the assignments unconditionally.
(...)	Parentheses group subexpressions. An operator, such as *, +, or [ ] works on a single character or on a regular expression enclosed in parentheses. For example, (a*(cb+)*)$0.

All of the preceding defined symbols are special. You must precede them with a \ (backslash) if you want to match the special symbol itself. For example, \$ matches a dollar sign.

Note: The **regcmp** subroutine uses the **malloc** subroutine to make the space for the vector. Always free the vectors that are not required. If you do not free the unneeded vectors, you can run out of memory if the **regcmp** subroutine is called repeatedly. Use the following as a replacement for the **malloc** subroutine to reuse the same vector, thus saving time and space:

```
/*  . . . Your Program . . .  */
malloc(n)
   int n;
{
   static int rebuf[256] ;

 return ((n <= sizeof(rebuf)) ? rebuf : NULL);
}
```

The **regcmp** subroutine produces code values that the **regex** subroutine can interpret as the regular expression. For instance, [a-z] indicates a range expression which the **regcmp** subroutine compiles into a string containing the two end points (a and z).

The **regex** subroutine interprets the range statement according to the current collating sequence. The expression [a-z] can be equivalent either to [abcd . . . xyz] , or to [aBbCcDd . . . xXyYzZ], as long as the character *preceding* the minus sign has a lower collating value than the character *following* the minus sign.

The behavior of a range expression is dependent on the collation sequence. If you want to match a *specific* set of characters, you should list each one. For example, to select letters a, b, or c, use [abc] rather than [a-c] .

Note:

1. No assumptions are made at compile time about the actual characters contained in the range.
2. Do not use multibyte characters.
3. You can use the] (right bracket) itself within a pair of brackets if it immediately follows the leading [(left bracket) or [^ (a left bracket followed immediately by a circumflex).
4. You can also use the minus sign (or hyphen) if it is the first or last character in the expression. For example, the expression [ ] -0] matches either the right bracket ( ]), or the characters - through 0.

Parameters

Item	Description
Subject	Specifies a comparison string.
String	Specifies the *Pattern* to be compiled.
Pattern	Specifies the expression to be compared.
ret	Points to an address at which to store comparison data. The **regex** subroutine allows multiple **ret** *String* parameters.

regcomp Subroutine

Purpose

Compiles a specified basic or extended regular expression into an executable string.

Library

Standard C Library (**libc. a**)

Syntax

```
#include <regex.h>

int regcomp ( Preg,  Pattern,  CFlags)
const char *Preg;
const char *Pattern;
int CFlags;
```

Description

The **regcomp** subroutine compiles the basic or extended regular expression specified by the *Pattern* parameter and places the output in the structure pointed to by the *Preg* parameter.

Parameters

Item	Description
Preg	Specifies the structure to receive the compiled output of the **regcomp** subroutine.
Pattern	Contains the basic or extended regular expression to be compiled by the **regcomp** subroutine. The default regular expression type for the *Pattern* parameter is a basic regular expression. An application can specify extended regular expressions with the **REG_EXTENDED** flag. The maximum number of subexpressions in an extended regular expression is 23.
CFlags	Contains the bitwise inclusive OR of 0 or more flags for the **regcomp** subroutine. These flags are defined in the **regex.h** file: **REG_EXTENDED** Uses extended regular expressions. The maximum number of subexpressions in an extended regular expression is 23. **REG_ICASE** Ignores case in match. **REG_NOSUB** Reports only success or failure in the **regexec** subroutine. If this flag is not set, the **regcomp** subroutine sets the **re_nsub** structure to the number of parenthetic expressions found in the *Pattern* parameter. **REG_NEWLINE** Prohibits . (period) and nonmatching bracket expression from matching a new-line character. The ^ (circumflex) and $ (dollar sign) will match the zero-length string immediately following or preceding a new-line character.

Return Values

If successful, the **regcomp** subroutine returns a value of 0. Otherwise, it returns another value indicating the type of failure, and the content of the *Preg* parameter is undefined.

Error Codes

The following macro names for error codes may be written to the **errno** global variable under error conditions:

Item	Description
REG_BADPAT	Indicates a basic or extended regular expression that is not valid.
REG_ECOLLATE	Indicates a collating element referenced that is not valid.
REG_ECTYPE	Indicates a character class-type reference that is not valid.
REG_EESCAPE	Indicates a trailing \ in pattern.
REG_ESUBREG	Indicates a number in `\digit` is not valid or in error.
REG_EBRACK	Indicates a `[]` imbalance.
REG_EPAREN	Indicates a `\(\)` or `()` imbalance.
REG_EBRACE	Indicates a `\{\}` imbalance.
REG_BADBR	Indicates the content of `\{\}` is unusable: not a number, number too large, more than two numbers, or first number larger than second.
REG_ERANGE	Indicates an unusable end point in range expression.
REG_ESPACE	Indicates out of memory.
REG_BADRPT	Indicates a ? (question mark), * (asterisk), or + (plus sign) not preceded by valid basic or extended regular expression.

If the **regcomp** subroutine detects an illegal basic or extended regular expression, it can return either the **REG_BADPAT** error code or another that more precisely describes the error.

Examples

The following example illustrates how to match a string (specified in the *string* parameter) against an extended regular expression (specified in the *Pattern* parameter):

```
#include <sys/types.h>
#include <regex.h>
int
match(char *string, char *pattern)
{
   int     status;
   regex_t re;
```

```
   if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0)  {
           return(0) ;              /* report error */
   }
   status = regexec(&re, string, (size_t) 0, NULL, 0);
   regfree(&re);
   if (status != 0)  {
           return(0) ;              /* report error */
   }
   return(1);
}
```

In the preceding example, errors are treated as no match. When there is no match or error, the calling process can get details by calling the **regerror** subroutine.

regerror Subroutine

Purpose

Returns a string that describes the *ErrCode* parameter.

Library

Standard C Library (**libc. a**)

Syntax

```
#include <regex.h>

size_t regerror (ErrCode, Preg, ErrBuf, ErrBuf_Size)
int  ErrCode;
const regex_t * Preg;
char * ErrBuf;
size_t  ErrBuf_Size;
```

Description

The **regerror** subroutine provides a mapping from error codes returned by the **regcomp** and **regexec** subroutines to printable strings. It generates a string corresponding to the value of the *ErrCode* parameter, which is the last nonzero value returned by the **regcomp** or **regexec** subroutine with the given value of the *Preg* parameter. If the *ErrCode* parameter is not such a value, the content of the generated string is unspecified. The string generated is obtained from the **regex.cat** message catalog.

If the *ErrBuf_Size* parameter is not 0, the **regerror** subroutine places the generated string into the buffer specifier by the *ErrBuf* parameter, whose size in bytes is specified by the *ErrBuf_Size* parameter. If the string (including the terminating null character) cannot fit in the buffer, the **regerror** subroutine truncates the string and null terminates the result.

Parameters

Item	Description
ErrCode	Specifies the error for which a description string is to be returned.
Preg	Specifies the structure that holds the previously compiled output of the **regcomp** subroutine.
ErrBuf	Specifies the buffer to receive the string generated by the **regerror** subroutine.
ErrBuf_Size	Specifies the size of the *ErrBuf* parameter.

Return Values

The **regerror** subroutine returns the size of the buffer needed to hold the entire generated string, including the null termination. If the return value is greater than the value of the *ErrBuf_Size* variable, the string returned in the *ErrBuf* buffer is truncated.

Error Codes

If the *ErrBuf_Size* value is 0, the regerror subroutine ignores the *ErrBuf* parameter, but returns the one of the following error codes. These error codes defined in the regex.h file.

Item	Description
REG_NOMATCH	Indicates the basic or extended regular expression was unable to find a match.
REG_BADPAT	Indicates a basic or extended regular expression that is not valid.
REG_ECOLLATE	Indicates a collating element referenced that is not valid.
REG_ECTYPE	Indicates a character class-type reference that is not valid.
REG_EESCAPE	Indicates a trailing \ in pattern.
REG_ESUBREG	Indicates a number in `\digit` is not valid or in error.
REG_EBRACK	Indicates a `[]` imbalance.
REG_EPAREN	Indicates a `\(\)` or `()` imbalance.
REG_EBRACE	Indicates a `\{\}` imbalance.
REG_BADBR	Indicates the content of `\{\}` is unusable: not a number, number too large, more than two numbers, or first number larger than second.
REG_ERANGE	Indicates an unusable end point in range expression.
REG_ESPACE	Indicates out of memory.
REG_BADRPT	Indicates a ? (question mark), * (asterisk), or + (plus sign) not preceded by valid basic or extended regular expression.
REG_ENEWLINE	Indicates a new-line character was found before the end of the regular or extended regular expression, and **REG_NEWLINE** was not set.

If the *Preg* parameter passed to the **regexec** subroutine is not a compiled basic or extended regular expression returned by the **regcomp** subroutine, the result is undefined.

Examples

An application can use the **regerror** subroutine (with the parameters **(***Code*, *Preg*, null, (**size_t**) **0)** passed to it) to determine the size of buffer needed for the generated string, call the **malloc** subroutine to allocate a buffer to hold the string, and then call the **regerror** subroutine again to get the string. Alternately, this subroutine can allocate a fixed, static buffer that is large enough to hold most strings (perhaps 128 bytes), and then call the **malloc** subroutine to allocate a larger buffer if necessary.

regexec Subroutine

Purpose

Compares the null-terminated string specified by the value of the *String* parameter against the compiled basic or extended regular expression *Preg*, which must have previously been compiled by a call to the **regcomp** subroutine.

Library

Standard C Library (**libc. a**)

Syntax

```
#include <regex.h>

int regexec (Preg, String, NMatch, PMatch, EFlags)
const regex_t * Preg;
const char * String;
```

```
size_t  NMatch;
regmatch_t * PMatch;
int  EFlags;
```

Description

The **regexec** subroutine compares the null-terminated string in the *String* parameter with the compiled basic or extended regular expression in the *Preg* parameter initialized by a previous call to the **regcomp** subroutine. If a match is found, the **regexec** subroutine returns a value of 0. The **regexec** subroutine returns a nonzero value if it finds no match or it finds an error.

If the *NMatch* parameter has a value of 0, or if the **REG_NOSUB** flag was set on the call to the **regcomp** subroutine, the **regexec** subroutine ignores the *PMatch* parameter. Otherwise, the *PMatch* parameter points to an array of at least the number of elements specified by the *NMatch* parameter. The **regexec** subroutine fills in the elements of the array pointed to by the *PMatch* parameter with offsets of the substrings of the *String* parameter. The offsets correspond to the parenthetic subexpressions of the original *pattern* parameter that was specified to the **regcomp** subroutine.

The **pmatch.rm_so** structure is the byte offset of the beginning of the substring, and the **pmatch.rm_eo** structure is one greater than the byte offset of the end of the substring. Subexpression *i* begins at the *i* th matched open parenthesis, counting from 1. The 0 element of the array corresponds to the entire pattern. Unused elements of the *PMatch* parameter, up to the value *PMatch*[*NMatch*-1], are filled with -1. If more than the number of subexpressions specified by the *NMatch* parameter (the *pattern* parameter itself counts as a subexpression), only the first *NMatch*-1 subexpressions are recorded.

When a basic or extended regular expression is being matched, any given parenthetic subexpression of the *pattern* parameter might match several different substrings of the *String* parameter. Otherwise, it might not match any substring even though the pattern as a whole did match.

The following rules are used to determine which substrings to report in the *PMatch* parameter when regular expressions are matched:

- If a subexpression in a regular expression participated in the match several times, the offset of the last matching substring is reported in the *PMatch* parameter.
- If a subexpression did not participate in a match, the byte offset in the *PMatch* parameter is a value of -1. A subexpression does not participate in a match if any of the following are true:
  - An * (asterisk) or \{\} (backslash, left brace, backslash, right brace) appears immediately after the subexpression in a basic regular expression.
  - An * (asterisk), ? (question mark), or { } (left and right braces) appears immediately after the subexpression in an extended regular expression and the subexpression did not match (matched 0 times).
  - A | (pipe) is used in an extended regular expression to select either the subexpression that didn't match or another subexpression, and the other subexpression matched.
- If a subexpression is contained in a subexpression, the data in the *PMatch* parameter refers to the last such subexpression.
- If a subexpression is contained in a subexpression and the byte offsets in the *PMatch* parameter have a value of -1, the pointers in the *PMatch* parameter also have a value of -1.
- If a subexpression matched a zero-length string, the offsets in the *PMatch* parameter refer to the byte immediately following the matching string.

If the **REG_NOSUB** flag was set in the *cflags* parameter in the call to the **regcomp** subroutine, and the *NMatch* parameter is not equal to 0 in the call to the **regexec** subroutine, the content of the *PMatch* array is unspecified.

If the **REG_NEWLINE** flag was not set in the *cflags* parameter when the **regcomp** subroutine was called, then a new-line character in the *pattern* or *String* parameter is treated as an ordinary character. If the **REG_NEWLINE** flag was set when the **regcomp** subroutine was called, the new-line character is treated as an ordinary character except as follows:

- A new-line character in the *String* parameter is not matched by a period outside of a bracket expression or by any form of a nonmatching list. A nonmatching list expression begins with a ^ (circumflex) and specifies a list that matches any character or collating element and the expression in the list after the leading caret. For example, the regular expression `[^abc]` matches any character except a, b, or c. The circumflex has this special meaning only when it is the first character in the list, immediately following the left bracket.
- A ^ (circumflex) in the *pattern* parameter, when used to specify expression anchoring, matches the zero-length string immediately after a new-line character in the *String* parameter, regardless of the setting of the **REG_NOTBOL** flag.
- A $ (dollar sign) in the *pattern* parameter, when used to specify expression anchoring, matches the zero-length string immediately before a new-line character in the *String* parameter, regardless of the setting of the **REG_NOTEOL** flag.

Parameters

Item	Description
Preg	Contains the compiled basic or extended regular expression to compare against the *String* parameter.
String	Contains the data to be matched.
NMatch	Contains the number of subexpressions to match.
PMatch	Contains the array of offsets into the *String* parameter that match the corresponding subexpression in the *Preg* parameter.
EFlags	Contains the bitwise inclusive OR of 0 or more of the flags controlling the behavior of the **regexec** subroutine capable of customizing. The *EFlags* parameter modifies the interpretation of the contents of the *String* parameter. It is the bitwise inclusive OR of 0 or more of the following flags, which are defined in the **regex.h** file: **REG_NOTBOL** The first character of the string pointed to by the *String* parameter is not the beginning of the line. Therefore, the ^ (circumflex), when used as a special character, does not match the beginning of the *String* parameter. **REG_NOTEOL** The last character of the string pointed to by the *String* parameter is not the end of the line. Therefore, the $ (dollar sign), when used as a special character, does not match the end of the *String* parameter.

Return Values

On successful completion, the **regexec** subroutine returns a value of 0 to indicate that the contents of the *String* parameter matched the contents of the *pattern* parameter, or to indicate that no match occurred. The **REG_NOMATCH** error is defined in the **regex.h** file.

Error Codes

If the **regexec** subroutine is unsuccessful, it returns a nonzero value indicating the type of problem. The following macros for possible error codes that can be returned are defined in the **regex.h** file:

Item	Description
REG_NOMATCH	Indicates the basic or extended regular expression was unable to find a match.
REG_BADPAT	Indicates a basic or extended regular expression that is not valid.
REG_ECOLLATE	Indicates a collating element referenced that is not valid.

Item	Description
REG_ECTYPE	Indicates a character class-type reference that is not valid.
REG_EESCAPE	Indicates a trailing \ (backslash) in the pattern.
REG_ESUBREG	Indicates a number in `\digit` is not valid or is in error.
REG_EBRACK	Indicates a [ ] (left and right brackets) imbalance.
REG_EPAREN	Indicates a \ (\ ) (backslash, left parenthesis, backslash, right parenthesis) or ( ) (left and right parentheses) imbalance.
REG_EBRACE	Indicates a \ { \ } (backslash, left brace, backslash, right brace) imbalance.
REG_BADBR	Indicates the content of \ { \ } (backslash, left brace, backslash, right brace) is unusable (not a number, number too large, more than two numbers, or first number larger than second).
REG_ERANGE	Indicates an unusable end point in range expression.
REG_ESPACE	Indicates out of memory.
REG_BADRPT	Indicates a ? (question mark), * (asterisk), or + (plus sign) not preceded by valid basic or extended regular expression.

If the value of the *Preg* parameter to the **regexec** subroutine is not a compiled basic or extended regular expression returned by the **regcomp** subroutine, the result is undefined.

Examples

The following example demonstrates how the **REG_NOTBOL** flag can be used with the **regexec** subroutine to find all substrings in a line that match a pattern supplied by a user. (For simplicity, very little error-checking is done in this example.)

```
(void) regcomp (&re, pattern, 0) ;
/* this call to regexec finds the first match on the line */
error = regexec (&re, &buffer[0], 1, &pm, 0) ;
while (error = = 0) {   /* while matches found */
<subString found between pm.r._sp and pm.rm_ep>
/* This call to regexec finds the next match */
error = regexec (&re, pm.rm_ep, 1, &pm, REG_NOTBOL) ;
```

regfree Subroutine

Purpose

Frees any memory allocated by the **regcomp** subroutine associated with the *Preg* parameter.

Library

Standard C Library (**libc. a**)

Syntax

```
#include <regex.h>
```

void regfree (*Preg***)**
regex_t ****Preg*;**

Description

The **regfree** subroutine frees any memory allocated by the **regcomp** subroutine associated with the *Preg* parameter. An expression defined by the *Preg* parameter is no longer treated as a compiled basic or extended regular expression after it is given to the **regfree** subroutine.

Parameters

Item	Description
Preg	Structure containing the compiled output of the **regcomp** subroutine. Memory associated with this structure is freed by the **regfree** subroutine.

reltimerid Subroutine

Purpose

Releases a previously allocated interval timer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>
#include <sys/events.h>

int reltimerid ( TimerID)
timer_t TimerID;
```

Description

The **reltimerid** subroutine is used to release a previously allocated interval timer, which is returned by the **gettimerid** subroutine. Any pending timer event generated by this interval timer is cancelled when the call returns.

Parameters

Item	Description
TimerID	Specifies the ID of the interval timer being released.

Return Values

The **reltimerid** subroutine returns a 0 if it is successful. If an error occurs, the value -1 is returned and **errno** is set.

Error Codes

If the **reltimerid** subroutine fails, a -1 is returned and **errno** is set with the following error code:

Item	Description
EINVAL	The timer ID specified by the *Timerid* parameter is not a valid timer ID.

remainder, remainderf, remainderl, remainderd32, remainderd64, and remainderd128 Subroutines

Purpose

Returns the floating-point remainder.

Syntax

```
#include <math.h>

double remainder (x, y)
double x;
double y;

float remainderf (x, y)
float x;
float y;

long double remainderl (x, y)
long double x;
long double y ;
_Decimal32 remainderd32 (x, y)
_Decimal32 x;
_Decimal32 y;

_Decimal64 remainderd64 (x, y)
_Decimal64 x;
_Decimal64 y;

_Decimal128 remainderd128 (x, y)
_Decimal128 x;
_Decimal128 y;
```

Description

The **remainder**, **remainderf**, **remainderl**, **remainderd32**, **remainderd64**, and **remainderd128** subroutines return the floating-point remainder *r=x - ny* when *y* is nonzero. The value *n* is the integral value nearest the exact value *x/y*. When | *n x/y* |=½ , the value *n* is chosen to be even.

Parameters

Item	Description
x	Specifies the value of the numerator.
y	Specifies the value of the denominator.

Return Values

Upon successful completion, the **remainder**, **remainderf**, **remainderl**, **remainderd32**, **remainderd64**, and **remainderd128** subroutines return the floating-point remainder *r=x - ny* when *y* is nonzero.

If *x* or *y* is NaN, a NaN is returned.

If *x* is infinite or *y* is 0 and the other is non-NaN, a domain error occurs, and a NaN is returned.

remove Subroutine

Purpose

Removes a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>

int remove( FileName)
const char *FileName;
```

Description

The **remove** subroutine makes a file named by *FileName* inaccessible by that name. An attempt to open that file using that name does not work unless you recreate it. If the file is open, the subroutine does not remove it.

If the file designated by the *FileName* parameter has multiple links, the link count of files linked to the removed file is reduced by 1.

Parameters

Item	Description
FileName	Specifies the name of the file being removed.

Return Values

Upon successful completion, the **remove** subroutine returns a value of 0; otherwise it returns a nonzero value.

removeea Subroutine

Purpose

Removes an extended attribute.

Syntax

```
#include <sys/ea.h>

int removeea(const char *path, const char *name);
int fremoveea(int filedes, const char *name);
int lremoveea(const char *path, const char *name);
```

Description

Extended attributes are name:value pairs associated with the file system objects (such as files, directories, and symlinks). They are extensions to the normal attributes that are associated with all objects in the file system (that is, the **stat(2)** data).

Do not define an extended attribute name with the 8-character prefix "(0xF8)SYSTEM(0xF8)". Prefix "(0xF8)SYSTEM(0xF8)" is reserved for system use only.

Note: 0xF8 represents a non-printable character.

The **removeea** subroutine removes the extended attribute identified by *name* and associated with the given *path* in the file system. The **fremoveea** subroutine is identical to **removeea**, except that it takes a file descriptor instead of a path. The **lremoveea** subroutine is identical to **removeea**, except, in the case of a symbolic link, the link itself is interrogated rather than the file that it refers to.

Parameters

Item	Description
path	The path name of the file.
name	The name of the extended attribute. An extended attribute name is a NULL-terminated string.
filedes	A file descriptor for the file.

Return Values

If the **removeea** subroutine succeeds, 0 is returned. Upon failure, -1 is returned and **errno** is set appropriately.

Error Codes

Item	Description
EACCES	Caller lacks write permission on the base file, or lacks the appropriate ACL privileges for named attribute **delete**.
EFAULT	A bad address was passed for *path* or *name*.
EFORMAT	File system is capable of supporting EAs, but EAs are disabled.
EINVAL	A path-like name should not be used (such as **zml/file**, **.** and **..**).
ENOATTR	The named attribute does not exist, or tthe process has no access to this attribute.
ENOTSUP	Extended attributes are not supported by the file system.

remquo, remquof, remquol, remquod32, remquod64, and remquod128 Subroutines

Purpose

Returns the floating-point remainder.

Syntax

```
#include <math.h>

double remquo (x, y, quo)
double x;
double y;
int *quo;

float remquof (x, y, quo)
float x;
float y;
int *quo;

long double remquol (x, y, quo)
long double x;
long double y;
int *quo;
_Decimal32 remquod32 (x, y, quo)
_Decimal32 x;
_Decimal32 y;
int *quo;

_Decimal64 remquod64 (x, y, quo)
_Decimal64 x;
_Decimal64 y;
```

```
int *quo;

_Decimal128 remquod128 (x, y, quo)
_Decimal128 x;
_Decimal128 y;
int *quo;
```

Description

The **remquo**, **remquof**, **remquol**, **remquod32**, **remquod64**, **remquod128** subroutines compute the same remainder as the **remainder**, **remainderf**, **remainderl remainderd32**, **remainderd64**, and **remainder128** functions, respectively. In the object pointed to by *quo*, they store a value whose sign is the sign of *x/y* and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of *x/y*, where *n* is 3.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value of the numerator.
y	Specifies the value of the denominator.
quo	Points to the object where a value whose sign is the sign of *x/y* is stored.

Return Values

The **remquo**, **remquof**, **remquol**, **remquod32**, **remquod64**, and **remquod128** subroutines return *x* REM *y*.

If *x* or *y* is NaN, a NaN is returned.

If *x* is ±Inf or *y* is zero and the other argument is non-NaN, a domain error occurs, and a NaN is returned.

rename or renameat Subroutine

Purpose

Renames a directory or a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>

int rename ( FromPath,  ToPath)
const char *FromPath, *ToPath;

int renameat (DirFileDescriptor1, FromPath, DirFileDescriptor2,
ToPath)
int DirFileDescriptor1, DirFileDescriptor2;
const char *FromPath, *ToPath;
```

Description

The **rename** and **renameat** subroutines rename a directory or a file within a file system. The **renameat** subroutine is equivalent to the rename subroutine if both *DirFileDescriptor1* and *DirFileDescriptor2* are **AT_FDCWD** or both *the FromPath* and *ToPath* parameters are absolute path names.

To use either subroutine, the calling process must have write and search permission in the parent directories of both the *FromPath* and *ToPath* parameters. If either directory pointed at by the *DirFileDescriptor1* or *DirFileDescriptor2* parameter in the **renameat** subroutine was opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted for that directory using the current permissions of the directory. However, if either directory was opened with the **O_SEARCH** open flag, the subroutine does not perform the check for that directory. If the path defined in the *FromPath* parameter is a directory, the calling process must have write and search permission to the *FromPath* directory as well. If both the *FromPath* and *ToPath* parameters refer to the same existing file, both subroutines return successfully and perform no other action.

The components of both the *FromPath* and *ToPath* parameters must be of the same type (that is, both directories or both non-directories) and must reside on the same file system. If the *ToPath* file already exists, it is first removed. Removing it guarantees that a link named *ToPath* will exist throughout the operation. This link refers to the file named by either the *ToPath* or *FromPath* parameter before the operation began.

If the final component of the *FromPath* parameter is a symbolic link, the symbolic link (not the file or directory to which it points) is renamed. If the *ToPath* is a symbolic link, the link is destroyed.

If the parent directory of the *FromPath* parameter has the Sticky bit attribute (described in the **<sys/mode.h>** file), the calling process must have an effective user ID equal to the owner ID of the *FromPath* parameter, or to the owner ID of the parent directory of the *FromPath* parameter.

A user who is not the owner of the file or directory must have root user authority to use the **rename** subroutine.

If the *FromPath* and *ToPath* parameters name directories, the following must be true:

- The directory specified by the *FromPath* parameter is not an ancestor of *ToPath.* For example, the *FromPath* path name must not contain a path prefix that names the directory specified by the *ToPath* parameter.
- The directory specified in the *FromPath* parameter must be well-formed. A well-formed directory contains both . (dot) and .. (dot dot) entries. That is, the . (dot) entry in the *FromPath* directory refers to the same directory as that in the *FromPath* parameter. The .. (dot dot) entry in the *FromPath* directory refers to the directory that contains an entry for *FromPath*.
- The directory specified by the *ToPath* parameter, if it exists, must be well-formed (as defined previously).

Parameters

Item	Description
DirFileDescriptor1	Specifies the file descriptor of an open directory.
DirFileDescriptor2	Specifies the file descriptor of an open directory.
FromPath	Identifies the file or directory to be renamed. If *DirFileDescriptor1* is specified and *FromPath* is a relative path name, then *FromPath* is considered relative to the directory specified by *DirFileDescriptor1*.
ToPath	Identifies the new path name of the file or directory to be renamed. If *DirFileDescriptor2* is specified and *ToPath* is a relative path name, then *ToPath* is considered relative to the directory specified by *DirFileDescriptor2*. If *ToPath* is an existing file or empty directory, it is replaced by *FromPath*. If *ToPath* specifies a directory that is not empty, the **rename** subroutine exits with an error.

Return Values

Upon successful completion, the **rename** and **renameat** subroutines return a value of 0. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **rename** or **renameat** subroutine is unsuccessful and the file or directory name remains unchanged if one or more of the following are true:

Item	Description
EACCES	Creating the requested link requires writing in a directory mode that denies the process write permission.
EBUSY	The directory named by the *FromPath* or *ToPath* parameter is currently in use by the system, or the file named by *FromPath* or *ToPath* is a named STREAM.
EDQUOT	The directory that would contain the path specified by the *ToPath* parameter cannot be extended because the user's or group's quota of disk blocks on the file system containing the directory is exhausted.
EEXIST	The *ToPath* parameter specifies an existing directory that is not empty.
EINVAL	The path specified in the *FromPath* or *ToPath* parameter is not a well-formed directory (*FromPath* is an ancestor of *ToPath*), or an attempt has been made to rename . (dot) or .. (dot dot).
EISDIR	The *ToPath* parameter names a directory and the *FromPath* parameter names a non-directory.
EMLINK	The *FromPath* parameter names a directory that is larger than the maximum link count of the parent directory of the *ToPath* parameter.
ENOENT	A component of either path does not exist, the file named by the *FromPath* parameter does not exist, or a symbolic link was named, but the file to which it refers does not exist.
ENOSPC	The directory that would contain the path specified in the *ToPath* parameter cannot be extended because the file system is out of space.
ENOTDIR	The *FromPath* parameter names a directory and the *ToPath* parameter names a non-directory.
ENOTEMPTY	The *ToPath* parameter specifies an existing directory that is not empty.
EROFS	The requested operation requires writing in a directory on a read-only file system.
ETXTBSY	The *ToPath* parameter names a shared text file that is currently being used.
EXDEV	The link named by the *ToPath* parameter and the file named by the *FromPath* parameter are on different file systems.

The **renameat** subroutine is unsuccessful and the file or directory name remains unchanged if one or more of the following are true:

Item	Description
EACCES	The directory pointed at by the *DirFileDescriptor1* or *DirFileDescriptor2* parameter was not opened with the **O_SEARCH** flag and the permissions of the directory do not permit directory searches.
EBADF	A *Path* parameter does not specify an absolute path and the corresponding *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a valid file descriptor.

Item	Description
ENOTDIR	A *Path* parameter does not specify an absolute path and the corresponding *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.

If Network File System (NFS) is installed on the system, the **rename** and **renameat** subroutines can be unsuccessful if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

The **rename** and **renameat** subroutines can be unsuccessful for other reasons.

reset_malloc_log Subroutine

Purpose

Resets information collected by the malloc subsystem.

Syntax

```
#include <malloc.h>
void reset_malloc_log (addr)
void *addr;
```

Description

The **reset_malloc_log** subroutine resets the record of currently active malloc allocations stored by the malloc subsystem. These records are stored in **malloc_log** structures, which are located in the process heap. Only records corresponding to the heap of which *addr* is a member are reset, unless *addr* is NULL, in which case records for all heaps are reset. The *addr* parameter must be a pointer to space allocated previously by the malloc subsystem or NULL, otherwise no information is reset and the **errno** global variable is set to **EINVAL**.

Parameters

Item	Description
addr	Pointer to space allocated previously by the malloc subsystem

reset_prog_mode Subroutine

Purpose

Restores the terminal to program mode.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
reset_prog_mode( )
```

Description

The **reset_prog_mode** subroutine restores the terminal to program or *in curses* mode.

The **reset_prog_mode** subroutine is a low-level routine and normally would not be called directly by a program.

reset_shell_mode Subroutine

Purpose

Restores the terminal to shell mode.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
reset_shell_mode( )
```

Description

The **reset_shell_mode** subroutine restores the terminal into shell , or "out of curses," mode. This happens automatically when the **endwin** subroutine is called.

resetterm Subroutine

Purpose

Resets terminal modes to what they were when the **saveterm** subroutine was last called.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
resetterm( )
```

Description

The **resetterm** subroutine resets terminal modes to what they were when the **saveterm** subroutine was last called.

The **resetterm** subroutine is called by the **endwin** subroutine, and should normally not be called directly by a program.

resetty, savetty Subroutine

Purpose

Saves/restores the terminal mode.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int resetty(void);

int savetty(void):
```

Description

The **resetty** subroutine restores the program mode as of the most recent call to the **savetty** subroutine.

The **savetty** subroutine saves the state that would be put in place by a call to the **reset_prog_mode** subroutine.

Return Values

Upon successful completion, these subroutines return OK. Otherwise. they return ERR.

Examples

To restore the terminal to the state it was in at the last call to **savetty**, enter:

```
resetty();
```

restartterm Subroutine

Purpose

Re-initializes the terminal structures after a restore.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
#include <term.h>
```

restartterm (*Term*, *FileNumber*, *ErrorCode*)
char ***Term**;
int *FileNumber*;
int ***ErrorCode**;

Description

The **restartterm** subroutine is similar to the **setupterm** subroutine except that it is called after restoring memory to a previous state. For example, you would call the **restartterm** subroutine after a call to **scr_restore** if the terminal type has changed. The **restartterm** subroutine assumes that the windows and the input and output options are the same as when memory was saved, but the terminal type and baud rate may be different.

Parameters

Item	Description
Term	Specifies the terminal name to obtain the terminal for. If 0 is passed for the parameter, the value of the **$TERM** environment variable is used.
FileNumber	Specifies the output file's file descriptor (1 equals standard out).
ErrorCode	Specifies a pointer to an integer to return the error code to. If 0, then the **restartterm** subroutine exits with an error message instead of returning.

Example

To restart an **aixterm** after a previous memory save and exit on error with a message, enter:

```
restartterm("aixterm", 1, (int*)0);
```

Prerequisite Information

Curses Overview for Programming and Understanding Terminals with Curses in *General Programming Concepts: Writing and Debugging Programs* .

revoke Subroutine

Purpose

Revokes access to a file.

Library

Standard C Library (**libc.a**)

Syntax

```
int revoke ( Path)
char *Path;
```

Description

The **revoke** subroutine revokes access to a file by all processes.

All accesses to the file are revoked. Subsequent attempts to access the file using a file descriptor established before the **revoke** subroutine fail and cause the process to receive a return value of -1, and the **errno** global variable is set to **EBADF**.

A process can revoke access to a file only if its effective user ID is the same as the file owner ID, or if the calling process is privileged.

Note: The **revoke** subroutine has no affect on subsequent attempts to open the file. To assure exclusive access to the file, the caller should change the access mode of the file before issuing the **revoke**

subroutine. Currently the **revoke** subroutine works only on terminal devices. The **chmod** subroutine changes file access modes.

Parameters

Item	Description
Path	Path name of the file for which access is to be revoked.

Return Values

Upon successful completion, the **revoke** subroutine returns a value of 0.

If the **revoke** subroutine fails, a value of -1 returns and the **errno** global variable is set to indicate the error.

Error Codes

The **revoke** subroutine fails if any of the following are true:

Item	Description
ENOTDIR	A component of the path prefix is not a directory.
EACCES	Search permission is denied on a component of the path prefix.
ENOENT	A component of the path prefix does not exist, or the process has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The path name is null.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ESTALE	The process's root or current directory is located in a virtual file system that has been unmounted.
EFAULT	The *Path* parameter points outside of the process's address space.
ELOOP	Too many symbolic links were encountered in translating the path name.
ENAMETOOLONG	A component of a path name exceeds 255 characters, or an entire path name exceeds 1023 characters.
EIO	An I/O error occurred during the operation.
EPERM	The effective user ID of the calling process is not the same as the file's owner ID.
EINVAL	Access rights revocation is not implemented for this file.

rintf, rintl, rint, rintd32, rintd64, or rintd128 Subroutine

Purpose

Rounds to the nearest integral value.

Syntax

```
#include <math.h>

float rintf (x)
float x;
```

```
long double rintl (x)
long double x;

double rint (x)
double x;

_Decimal32 rintd32(x)
_Decimal32 x;

_Decimal64 rintd64(x)
_Decimal64 x;

_Decimal128 rintd128(x)
_Decimal128 x;
```

Description

The **rintf**, **rintl**, **rint**, **rintd32**, **rintd64**, and **rintd128** subroutines return the integral value (represented as a floating-point number) nearest *x* in the direction of the current rounding mode. The current rounding mode is implementation-defined.

The **rintf**, **rintl**, **rint**, **rintd32**, **rintd64**, and **rintd128** subroutines differ from the **nearbyint**, **nearbyintf**, **nearbyintl**, **nearbyintd32**, **nearbyintd64**, and **nearbyintd128** subroutines only in that they may raise the inexact floating-point exception if the result differs in value from the argument.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be rounded.

Return Values

Upon successful completion, the **rintf**, **rintl**, **rint**, **rintd32**, **rintd64**, and **rintd128** subroutines return the integer (represented as a floating-point number) nearest *x* in the direction of the current rounding mode.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

If the correct value would cause overflow, a range error occurs the **rintf**, **rintl**, **rint**, **rintd32**, **rintd64**, and **rintd128** subroutines return the value of the macro ±**HUGE_VALF**, ±**HUGE_VALL**, ±**HUGE_VAL**, ±**HUGE_VAL_D32**, ±**HUGE_VAL_D64**, and ±**HUGE_VAL_D128** (with the same sign as *x*), respectively.

ripoffline Subroutine

Purpose

Reserves a line for a dedicated purpose.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include
<curses.h>
```

```
int
ripoffline(int line,
int (*init)(WINDOW *win,
int columns));
```

Description

The **ripoffline** subroutine reserves a screen line for use by the application.

Any call to the **ripoffline** subroutine must precede the call to the **initscr** or **newterm** subroutine. If line is positive, one line is removed from the beginning of stdstr; if line is negative, one line is removed from the end. Removal occurs during the subsequent call to the **initscr** or **newterm** subroutine. When the subsequent call is made, the subroutine pointed to by *init* is called with two arguments: a WINDOW pointer to the one-line window that has been allocated and an integer with the number of columns in the window. The initialisation subroutine cannot use the LINES and COLS external variables and cannot call the **wrefresh** or **doupdate** subroutine, but may call the **wnoutrefresh** subroutine.

Up to five lines can be ripped off. Calls to the **ripoffline** subroutine above this limit have no effect, but report success.

Parameters

Item	Description
line	
**init*	
columns	
**win*	

Return Values

The **ripoffline** subroutine returns OK.

Example

To remove three lines from the top of the screen, enter:

```
#include <curses.h>
```

```
ripoffline(1,initfunc);
ripoffline(1,initfunc);
ripoffline(1,initfunc);
```

```
initscr();
```

rmdir Subroutine

Purpose

Removes a directory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

int rmdir (*Path*)
const char **Path*;

Description

The **rmdir** subroutine removes the directory specified by the *Path* parameter. If Network File System (NFS) is installed on your system, this path can cross into another node.

For the **rmdir** subroutine to execute successfully, the calling process must have write access to the parent directory of the *Path* parameter.

In addition, if the parent directory of *Path* has the Sticky bit attribute (described in the **sys/mode.h** file), the calling process must have one of the following:

- An effective user ID equal to the directory to be removed
- An effective user ID equal to the owner ID of the parent directory of *Path*
- Root user authority.

Parameters

Item	Description
Path	Specifies the directory path name. The directory you specify must be: **Empty** The directory contains no entries other than . (dot) and .. (dot dot). **Well-formed** If the . (dot) entry in the *Path* parameter exists, it must refer to the same directory as *Path*. Exactly one directory has a link to the *Path* parameter, excluding the self-referential . (dot). If the .. (dot dot) entry in *Path* exists, it must refer to the directory that contains an entry for *Path*.

Return Values

Upon successful completion, the **rmdir** subroutine returns a value of 0. Otherwise, a value of -1 is returned, the specified directory is not changed, and the **errno** global variable is set to indicate the error.

Error Codes

The **rmdir** subroutine fails and the directory is not deleted if the following errors occur:

Item	Description
EACCES	There is no search permission on a component of the path prefix, or there is no write permission on the parent directory of the directory to be removed.
EBUSY	The directory is in use as a mount point.
EEXIST or **ENOTEMPTY**	The directory named by the *Path* parameter is not empty.
ENAMETOOLONG	The length of the *Path* parameter exceeds **PATH_MAX**; or a path-name component longer than **NAME_MAX** and **POSIX_NO_TRUNC** is in effect.
ENOENT	The directory named by the *Path* parameter does not exist, or the *Path* parameter points to an empty string.

Item	Description
ENOTDIR	A component specified by the *Path* parameter is not a directory.
EINVAL	The directory named by the *Path* parameter is not well-formed.
EROFS	The directory named by the *Path* parameter resides on a read-only file system.

If NFS is installed on the system, the **rmdir** subroutine fails if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

rmproj Subroutine

Purpose

Removes project definition from kernel project registry.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

rmproj(struct project *, int flag)
```

Description

The **rmproj** subroutine removes the definition of a project from kernel project registry. It takes a pointer to project structure as input argument that holds the name or number of a project that needs to be removed. The flag is set to indicate whether a name or number is supplied as input, as follows:

- PROJ_NAME — Indicates that the supplied project definition only has the project name. The **rmproj** subroutine queries the kernel to obtain a match for the supplied project name and returns the matching entry.
- PROJ_NUM — Indicates that the supplied project definition only has the project number. The **rmproj** subroutine queries the kernel to obtain a match for the supplied project number and returns the matching entry.

Parameters

Item	Description
project	Pointer holding the details of the project to be removed.
flag	An integer flag which indicates whether the supplied project definition structure has project name and number that need to be removed.

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
EINVAL	Pointer is null or the *flag* parameter is set to an invalid value.
ENOENT	Project Definition does not exist.
EPERM	Permission denied.

rmprojdb Subroutine

Purpose

Removes the specified project definition from the specified project database.

Library

The **libaacct.a** library.

Syntax

```
<sys/aacct.h>

rmprojdb(void *handle, struct project *project, int flag)
```

Description

The **rmprojdb** subroutine removes the project definition stored in the struct project variable from the project named by the *handle* parameter. The project database must be initialized before calling this subroutine. The **projdballoc** and **projdbfinit** subroutines are provided for this purpose. If the supplied project definition does not exist in the named project database, the **rmprojdb** subroutine returns -1 and sets errno to **ENOENT**.

The **rmprojdb** subroutine takes a pointer to a project structure as an input argument. This pointer to the project structure holds the name or number of a project that needs to be removed. The flag parameter is set to indicate whether a name or number is supplied as input as follows:

- PROJ_NAME — Indicates that the supplied project definition only has the project name.
- PROJ_NUM — Indicates that the supplied project definition only has the project number.

There is an internal state (that is, the current project) associated with the project database. When the project database is initialized, the current project is the first project in the database. The **rmprojdb** subroutine removes the named project and repositions the internal current project to the first project definition.

Parameters

Item	Description
handle	Pointer to project database handle.

Item	Description
project	Pointer to a project structure that holds the definition of the project to be added.
flag	Integer flag to indicated whether the name or number of the project is supplied.

Security

Only for privileged users. Privilege can be extended to nonroot users by granting the CAP_AACCT capability to a user.

Return Values

Item	Description
0	Success
-1	Failure

Error Codes

Item	Description
ENOENT	Project definition does not exist
EPERM	Permission denied. The user is not a privileged user.
EINVAL	Passed pointer is NULL or the *flag* parameter holds an invalid value.

round, roundf, roundl, roundd32, roundd64, or roundd128 Subroutine

Purpose

Rounds to the nearest integer value in a floating-point format.

Syntax

```
#include <math.h>

double round (x)
double x;

float roundf (x)
float x;

long double roundl (x)
long double x;

_Decimal32 roundd32(x)
_Decimal32 x;

_Decimal64 roundd64(x)
_Decimal64 x;

_Decimal128 roundd128(x)
_Decimal128 x;
```

Description

The **round**, **roundf**, **roundl**, **roundd32**, **roundd64**, and **roundd128** subroutines round the *x* parameter to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be rounded.

Return Values

Upon successful completion, the **round**, **roundf**, **roundl**, **roundd32**, **roundd64**, and **roundd128** subroutines return the rounded integer value.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

If the correct value would cause overflow, a range error occurs and the **round**, **roundf**, **roundl**, **roundd32**, **roundd64**, and **roundd128** subroutines return the value of the macro ±**HUGE_VAL**, ±**HUGE_VALF**, ±**HUGE_VALL**, ±**HUGE_VAL_D32**, ±**HUGE_VAL_D64** and ±**HUGE_VAL_D128** (with the same sign as *x*), respectively.

rpmatch Subroutine

Purpose

Determines whether the response to a question is affirmative or negative.

Library

Standard C Library (**libc. a**)

Syntax

```
#include <stdlib.h>

int rpmatch ( Response)
const char *Response;
```

Description

The **rpmatch** subroutine determines whether the expression in the *Response* parameter matches the affirmative or negative response specified by the **LC_MESSAGES** category in the current locale. Both expressions can be extended regular expressions.

Parameters

Item	Description
Response	Specifies input entered in response to a question that requires an affirmative or negative reply.

Return Values

This subroutine returns a value of 1 if the expression in the *Response* parameter matches the locale's affirmative expression. It returns a value of 0 if the expression in the *Response* parameter matches the locale's negative expression. If neither expression matches the expression in the *Response* parameter, a -1 is returned.

Examples

The following example shows an affirmative expression in the En_US locale. This example matches any expression in the *Response* parameter that begins with a y or Y followed by zero or more alphabetic characters, or it matches the letter o followed by the letter k.

```
^[yY][:alpha:]* | ok
```

RSiAddSetHot or RSiAddSetHotx Subroutine

Purpose

Add a single set of peer statistics to an already defined SpmiHotSet. .

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiHotVals *RSiAddSetHot(rhandle, HotSet, StatName,
GrandParent,
                                 maxresp, threshold, frequency, feed_type,
                                 except_type, severity, trap_no)
RSiHandle rhandle;
struct SpmiHotSet *HotSet;
char *StatName;
cx_handle GrandParent;
int maxresp;
int threshold;
int frequency;
int feed_type;
int excp_type;
int severity;
int trap_no;
```

```
struct SpmiHotVals *RSiAddSetHotx(rhandlex, HotSet, StatName,
GrandParent,
                                 maxresp, threshold, frequency, feed_type,
                                 except_type, severity, trap_no)
RSiHandlex rhandlex;
struct SpmiHotSet *HotSet;
char *StatName;
cx_handle GrandParent;
int maxresp;
int threshold;
int frequency;
int feed_type;
int excp_type;
int severity;
int trap_no;
```

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

HotSet
Specifies a pointer to a valid structure of type **SpmiHotSet** struct as created by the **RSiCreateHotSet or RSiCreateHotSet** subroutine call.

StatName
Specifies the name of the statistic within the subcontexts (peer contexts) of the context identified by the *GrandParent* parameter.

GrandParent
Specifies a valid **cx_handle** handle as obtained by another subroutine call. The handle must identify a context with at least one subcontext, which contains the statistic identified by the *StatName* parameter. If the context specified is one of the **RTime** contexts, no subcontext need to be created at the time the **SpmiAddSetHot** subroutine call is issued; the presence of the metric identified by the *StatName* parameter is checked against the context class description.

If the context specified has multiple levels of instantiable context below it (such as the **FS** and **RTime/ARM** contexts), the metric is searched only for the lowest context level. The **SpmiHotSet** created is a pseudo hotvals structure used to link together a peer group of **SpmiHotVals** structures, which are created under the covers, one for each subcontext of the *GrandParent* context. In the case of **RTime/ARM**, if additional contexts are later added under the *GrandParent* contexts, additional hotsets are added to the peer group. It is transparent to the application program, except that the **RSiGetHotItem, RSiGetHotItemx** subroutine call returns the peer group **SpmiHotVals** pointer rather than the pointer to the pseudo structure.

Note that specifying a specific volume group context (such as **FS/rootvg**) or a specific application context (such as **RTime/ARN/armpeek**) is still valid and won't involve creation of pseudo **SpmiHotVals** structures.

maxresp
Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. If specified as zero, indicates that all **SpmiHotItems** that meet the criteria specified by *threshold* must be returned, up-to a maximum of *maxresp* items. If both exceptions/traps and feeds are requested, the *maxresp* value is used to cap the number of exceptions/alerts as well as the number of items returned. If *feed_type* is specified as **SiHotAlways**, the *maxresp* parameter is still used to return at most *maxresp* items.

Where the *GrandParent* argument specifies a context that has multiple levels of instantiable contexts below it, the *maxresp* is applied to each of the lowest level contexts above the the actual peer contexts at a time. For example, if the *GrandParent* context is **FS** (file systems) and the system has three volume groups, then a *maxresp* value of 2 could cause up to a maximum of 2 x 3 = 6 responses to be generated.

threshold
Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. If specified as zero, indicates that all values read qualify to be returned in feeds. The value specified is compared to the data value read for each peer statistic. If the data value exceeds the *threshold*, it qualifies to be returned as an **SpmiHotItems** element in the **SpmiHotVals** structure. If the *threshold* is specified as a negative value, the value qualifies if it is lower than the numeric value of *threshold*. If *feed_type* is specified as **SiHotAlways**, the threshold value is ignored for feeds. For peer statistics of type **SiCounter**, the *threshold* must be specified as a rate per second; for **SiQuantity** statistics the *threshold* is specified as a level.

frequency
Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. Ignored for feeds. Specifies the minimum number of minutes that must expire between any two exceptions/traps generated from this `SpmiHotVals` structure. This value must be specified as no less than 5 minutes.

feed_type
Specifies if feeds of **SpmiHotItems** should be returned for this **SpmiHotVals** structure. The following values are valid:

SiHotNoFeed No feeds should be generated

SiHotThreshold Feeds are controlled by *threshold*.

SiHotAlways All values, up-to a maximum of *maxresp* must be returned as feeds.

excp_type
Controls the generation of exception data packets and/or the generation of SNMP Traps from **xmservd**. Note that these types of packets and traps can only actually be sent if **xmservd** is running. Because of this, exception packets and SNMP traps are only generated as long as **xmservd** is active. Traps can only be generated on AIX. The conditions for generating exceptions and traps are controlled by the *threshold* and *frequency* parameters. The following values are valid for *excp_type*:

SiNoHotException Generate neither exceptions not traps.

SiHotException Generate exceptions but not traps.

SiHotTrap Generate SNMP traps but not exceptions.

SiHotBoth Generate both exceptions and SNMP traps.

severity
Required to be positive and greater than zero if exceptions are generated, otherwise specify as zero. Used to assign a severity code to the exception for display by **exmon**.

trap_no
Required to be positive and greater than zero if SNMP traps are generated, otherwise specify as zero. Used to assign the trap number in the generated SNMP trap.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiHotVals**. If an error occurs, NULL is returned and an error text may be placed in the external character array **RSiEMsg**. If you attempt to add more values to a statset than the current local buffer size allows, **RSiErrno** is set to **RSiTooMany**. If you attempt to add more values than the buffer size of the remote host's **xmservd** daemon allows, **RSiErrno** is set to **RSiBadStat** and the status field in the returned packet is set to **too_many_values**.

The external integer **RSiMaxValues** holds the maximum number of values acceptable with the data-consumer's buffer size.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiChangeFeed or RSiChangeFeedx Subroutine

Purpose

Changes the frequency at which the **xmservd** on the host identified by the first argument daemon is sending **data_feed** packets for a statset.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiChangeFeed(rhandle, statset, msecs)
RSiHandle rhandle;struct SpmiStatSet *statset;int msecs;
```

```
int RSiChangeFeedx(rhandlex, statset, msecs)
RSiHandlex rhandlex;struct SpmiStatSet *statset;int msecs;
```

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

statset
Must be a pointer to a SpmiStatSet structure of type struct , which was previously returned by a successful **RSiCreateStatSet or RSiCreateStatSetx** subroutine call. Data feeding must have started for this **SpmiStatSet** structure through a previous **RSiStartFeed or RSiStartFeedx** subroutine call.

msecs
The number of milliseconds between the sending of **Hot_feed** packets. This number is rounded to a multiple of **min_remote_int** milliseconds by the **xmservd** daemon on the remote host. This minimum interval can be modified through the **-i** command line interval to **xmservd**.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Return Values

If successful, the subroutine returns zero, otherwise -1. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine's success or failure.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiChangeHotFeed or RSiChangeHotFeedx Subroutine

Purpose

Changes the frequency at which the **xmservd** on the host identified by the first argument daemon is sending **hot_feed** packets for a statset or checking if exceptions or SNMP traps should be generated.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiChangeFeed(rhandle, hotset, msecs)
RSiHandle rhandle;struct SpmiHotSet *hotset;intmsecs;
```

```
int RSiChangeFeedx(rhandlex, hotset, msecs)
RSiHandlex rhandlex;struct SpmiHotSet *hotset;int msecs;
```

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the RSiOpenx subroutine.

hotset
Must be a pointer to a `SpmiStatSet` structure of type struct, which was previously returned by a successful **RsiCreateHotSet or RsiCreateHotSetx** subroutine call. Data feeding must have started for the **SpmiHotSet** structure through a previous **RSiStartHotFeed or RSiStartHotFeedx** subroutine call.

msecs
The number of milliseconds between the sending of **Hot_feed** packets. This number is rounded to a multiple of **min_remote_int** milliseconds by the **xmservd** daemon on the remote host. This minimum interval can be modified through the **-i** command line interval to **xmservd**.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Return Values

If successful, the subroutine returns zero, otherwise -1. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine's success or failure.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiClose or RSiClosex Subroutine

Purpose

Terminates the Remote Statistic Interface (RSI) interface for a remote host connection.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
void RSiClose(rhandle)
RSiHandle rhandle;
```

```
void RSiClosex(rhandlex)
RSiHandlex rhandlex;
```

Description

The **RSiClose** subroutine is responsible for:

1. Removing the data-consumer program as a known data consumer on a particular host. This is done by sending a **going_down** packet to the host.
2. Marking the RSI handle as not active.
3. Releasing all memory allocated in connection with the RSI handle.
4. Terminating the RSI interface for a remote host.

A successful **RSiOpen or RSiOpenx** subroutine creates tables on the remote host it was issued against. Therefore, a data consumer program that has issued successful **RSiOpen** or **RSiOpenx** subroutine calls must issue an **RSiClose or RSiClosex** subroutine call for each **RSiOpen** or **RSiOpenx** call before the program exits so that the tables in the remote **xmservd** daemon can be released.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

The macro **RSiIsOpen** can be used to test whether an RSI handle is open. It takes an **RSiHandle** as argument and returns true (1) if the handle is open, otherwise false (0).

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiCreateHotSet or RSiCreateHotSetx Subroutine

Purpose

Creates an empty hotset on the remote host identified by the argument.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiHotSet *RSiCreateHotSet(rhandle)
RSiHandle rhandle;
```

```
struct SpmiHotSet *RSiCreateHotSetx(rhandlex)
RSiHandlex rhandlex;
```

Description

The **RSiCreateHotSet** subroutine allocates an SpmiStatSet structure. The structure is initialized as an empty **SpmiHotSet** and a pointer to the **SpmiHotSet** structure is returned.

The **SpmiHotSet** structure provides the anchor point to a set of peer statistics and must exist before the **RSiAddSetHot or RSiAddSetHotx** subroutine can be successfully called.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

Return Values

The **RSiCreateHotSet or RSiCreateHotSetx** subroutine returns a pointer to a structure of type **SpmiHotSet** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the **RSI**.

RSiCreateStatSet or RSiCreateStatSetx Subroutine

Purpose

Creates an empty statset on the remote host identified by the argument.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiStatSet *RSiCreateStatSet(rhandle)
RSiHandle rhandle;
```

```
struct SpmiStatSet *RSiCreateStatSetx(rhandlex)
RSiHandlex rhandlex;
```

Description

The **RSiCreateStatSet** subroutine allocates an SpmiStatSet structure. The structure is initialized as an empty **SpmiStatSet** and a pointer to the **SpmiStatSet** structure is returned.

The **SpmiStatSet** structure provides the anchor point to a set of statistics and must exist before the **"RSiPathAddSetStat or RSiPathAddSetStatx Subroutine" on page 1789RSiPathAddSetStat or RSiPathAddSetStatx** subroutine can be successfully called.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

Return Values

The **RSiCreateStatSet or RSiCreateStatSetx** subroutine returns a pointer to a structure of type **SpmiStatSet** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the **RSI**.

RSiDelSetHot or RSiDelSetHotx Subroutine

Purpose

Deletes a single set of peer statistics identified by an SpmiHotVals structure from an SpmiHotSet.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiDelSetHot(rhandle, hsp, hvp)
RSiHandle rhandle;struct SpmiHotSet *hsp;struct SpmiHotVals*hvp;
```

```
int RSiDelSetHotx(rhandlex, hsp, hvp)
RSiHandlex rhandlex;struct SpmiHotSet *hsp;struct SpmiHotVals*hvp;
```

Description

The **RSiDelSetHot** subroutine performs the following actions:

1. Validates that the **SpmiHotSet** structure identified by the second argument exists and contains the **SpmiHotVals** statistic identified by the third argument.
2. Deletes the **SpmiHotVals** value from the **SpmiHotSet** structure so that future **data_feed** packets do not include the deleted statistic.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

hsp
Must be a pointer to a SpmiHotSet structure of type struct , which was previously returned by a successful **RSiCreateHotSet or RSiCreateHotSetx** subroutine call.

hvp
Must be a handle of SpmiHotVals structure of type struct as returned by a successful **RSiAddSetHot or RSiAddSetHotx** subroutine call. You cannot specify an **SpmiHotVals** structure that was internally

generated by the Spmi library code as described under the *GrandParent* parameter to **RSiAddSetHot or RSiAddSetHotx**.

Return Values

If successful, the subroutine returns a zero value; otherwise it returns a non-zero value and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiDelSetStat or RSiDelSetStatx Subroutine

Purpose

Deletes a single statistic identified by an SpmiStatVals pointer from an SpmiStatSet.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiDelSetStat(rhandle, ssp, svp)
RSiHandle rhandle;struct SpmiStatSet *ssp;struct SpmiStatVals*svp;
```

```
int RSiDelSetStatx(rhandlex, ssp, svp)
RSiHandlex rhandlex;struct SpmiStatSet *ssp;struct SpmiStatVals*svp;
```

Description

The **RSiDelSetStat, RSiDelSetStatx** subroutines performs the following actions:

1. Validates the **SpmiStatSet** structure identified by the second argument exists and contains the **SpmiStatVals** statistic identified by the third argument.
2. Deletes the **SpmiStatVals** value from the **SpmiStatSet** structure so that future **data_feed** packets do not include the deleted statistic.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

ssp
Must be a pointer to a `SpmiStatSet` structure of type struct , which was previously returned by a successful **RSiCreateStatSet, RSiCreateStatSetx** subroutine call.

svp
Must be a handle of the `SpmiStatVals` structure of type struct as returned by a successful **RSiPathAddSetStat, RSiPathAddSetStatx** subroutine call.

Return Values

If successful, the subroutine returns a zero value; otherwise it returns a non-zero value and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiFirstCx or RSiFirstCxx Subroutine

Purpose

Returns the first subcontext of an `SpmiCx` context.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiCxLink *RSiFirstCx(rhandle, context, name,
descr)
RSiHandle rhandle;
cx_handle *context;
char **name;
char **descr;
```

```
struct SpmiCxLink *RSiFirstCxx(rhandlex, context, name,
descr)
RSiHandlex rhandlex;
cx_handle *context;
char **name;
char **descr;
```

Description

The **RSiFirstCx** subroutine performs the following actions:

1. Validates that the context identified by the second argument exists.
2. Returns a handle to the first element of the list of subcontexts defined for the context.
3. Returns the short name and description of the subcontext.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

context
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx, RSiPathGetCxx** subroutine call.

name
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the short name of the subcontext is returned in the character array pointer.

descr
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the description of the subcontext is returned in the character array pointer.

Return Values

If successful, the subroutine returns a pointer to a structure of type **struct Sprinkling**. If an error occurs or if the context doesn't contain subcontexts, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiFirstStat or RSiFirstStatx Subroutine

Purpose

Returns the first statistic of an SpmiCx context.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiStatLink *RSiFirstStat(rhandle, context, name,
descr)
RSiHandle rhandle;
cx_handle *context;
char **name;
char **descr;
```

```
struct SpmiStatLink *RSiFirstStatx(rhandlex, context, name,
descr)
RSiHandlex rhandlex;
cx_handle *context;
char **name;
char **descr;
```

Description

The **RSiFirstStat or RSiFirstStatx** subroutine performs the following actions:

1. Validates that the context identified by the second argument exists.
2. Returns a handle to the first element of the list of statistics defined for the context.
3. Returns the short name and description of the statistic.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

context
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx or RSiPathGetCxx** subroutine call.

name
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the short name of the subcontext is returned in the character array pointer.

descr
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the description of the subcontext is returned in the character array pointer.

Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiStatLink**. If an error occurs, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiGetCECData or RSiGetCECDatax Subroutine

Purpose

Request that `xmtopas` command send the central electronics complex (CEC) aggregation data.

Library

RSI library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
int RSiGetCECData (rsh, cec stats, node_stats);
RsiHandle rsh;
Cec_Stats **cec_stats;
Node_Stats **node_stats;
```

```
int RSiGetCECDatax (rshx, cec stats, node_stats);
RsiHandlex rshx;
Cec_Stats **cec_stats;
Node_Stats **node_stats;
```

Description

The **RSiGetCECData or RSiGetCECDatax** subroutine returns the Aggregated Statistics for a CEC and also returns the statistics of individual nodes of the same CEC. This routine allocates memory for CEC and node statistics data structures. The count of individual nodes is available in the Cec_Stats structure. If an error, the subroutine returns -1.

Parameters

rsh
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rshx
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

cec_stats
Must be a pointer to point to a structure of type **struct Cec_Stats**.

node_stats
Must be a pointer to point to a structure of type **struct Node_Stats**.

Return Values

If successful, the subroutine returns 0.

If an error occurs, the subroutine returns -1 and error text is placed in the RSiEMsg external character array.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- `extern char RSiEMsg[];`
- `extern int RSiErrno;`

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum RSiErrorType.

Files

/usr/include/sys/Rsi.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiGetClusterData or RSiGetClusterDatax Subroutine

Purpose

Request that `xmtopas` command send the cluster aggregation data.

Library

RSI library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
int RSiGetClusterData(rsh, cluster_stats, node_stats);
RsiHandle rsh;
Cluster_Stats **cluster_stats;
Node_Stats **node_stats;
```

```
int RSiGetClusterDatax (rshx, cluster_stats, node_stats);
RsiHandlex rshx;
Cluster_Stats **cluster_stats;
Node_Stats **node_stats;
```

Description

The **RSiGetClusterData or RSiGetClusterDatax** subroutine returns the Aggregated Statistics for a Cluster and also returns the statistics of individual nodes of the monitored cluster. This routine allocates memory for Cluster & Node statistics data structures. The count of individual nodes is available in the Cluster_Stats structure. If an error, the subroutine returns -1.

Parameters

rsh
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen ("RSiOpen or RSiOpenx Subroutine" on page 1787)** subroutine.

rshx
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

cluster_stats
Must be a pointer to point to a structure of type **struct Cluster_Stats**.

node_stats
Must be a pointer to point to a structure of type **struct Node_Stats**.

Return Values

If successful, the subroutine returns 0.

If an error occurs, the subroutine returns -1 and error text is placed in the RSiEMsg external character array.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- `extern char RSiEMsg[];`
- `extern int RSiErrno;`

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

/usr/include/sys/Rsi.h Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiGetHotItem or RSiGetHotItemx Subroutine

Purpose

Locates and decodes the next **SpmiHotItems** element at the current position in an incoming data packet of type **hot_feed**.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiHotVals *RSiGetHotItem(rhandle, HotSet, index, value,
absvalue, name)
RSiHandle rhandle;
struct SpmiHotSet **HotSet;
int *index;
float *value;
flost absvalue;
char **name;
```

```
struct SpmiHotVals *RSiGetHotItemx(rhandlex, HotSet, index, value,
absvalue, name)
RSiHandlex rhandlex;
struct SpmiHotSet **HotSet;
int *index;
float *value;
flost absvalue;
char **name;
```

Description

The **RSiGetHotItem** subroutine locates the **SpmiHotItems** structure in the **hot_feed** data packet indexed by the value of the *index* parameter. The subroutine returns a NULL value if no further **SpmiHotItems** structures are found. The **RSiGetHotItem** subroutine should only be executed after a successful call to the **RSiGetHotSet** subroutine.

The **RSiGetHotItem** subroutine is designed to be used for walking all **SpmiHotItems** elements returned in a **hot_feed** data packet. Because the data packet may contain elements belonging to more than one **SpmiHotSet**, the *index* is purely abstract and is only used to keep position. By feeding the updated integer pointed to by *index* back to the next call, the walking of the **hot_feed** packet can be done in a tight loop. Successful calls to **RSiGetHotItem or RSiGetHotItemx** subroutine decodes each **SpmiHotItems** element and return the data value in *value* and the name of the peer context that owns the corresponding statistic in *name*.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

HotSet
Used to return a pointer to a valid `SpmiHotSet` structure as obtained by a previous **RSiCreateHotSet or RSiCreateHotSetx** subroutine call. The calling program can use this value to locate the **SpmiHotSet** if its address was stored by the program after it was created. The time stamps in the **SpmiHotSet** are updated with the time stamps of the decoded **SpmiHotItems** element.

index
A pointer to an integer that contains the desired relative element number in the `SpmiHotItems` array across all `SpmiStatVals` contained in the data packet. A value of zero points to the first element. When the **RSiGetHotItem or RSiGetHotItemx** subroutine returns, the integer contain the index of the next **SpmiHotItems** element in the data packet. By passing the returned *index* parameter to the next call to **RSiGetHotItem or RSiGetHotItemx**, the calling program can iterate through all **SpmiHotItems** elements in the **hot_feed** data packet.

value
A pointer to a float variable. A successful call returns the decoded data value of the peer statistic. Before the value is returned, the **RSiGetHotItem or RSiGetHotItemx** function:

- Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing.
- Determines the data value as being either type **SiQuantity** or type **SiCounter** and performs one of the actions listed here:
  - If the data value is of type **SiQuantity**, the subroutine returns the **val** field of the **SpmiHotItems** structure.
  - If the data value is of type **SiCounter**, the subroutine returns the value of the **val_change** field of the **SpmiHotItems** structure divided by the elapsed number of seconds since the previous time a data value was requested for this set of statistics.

absvalue
A pointer to a float variable. A successful call will return the decoded value of the **val** field of the **SpmiHotItems** structure of the peer statistic. In case of a statistic of type **SiQuantity**, this value will be the same as the one returned in the argument *value*. In case of a peer statistic of type **SiCounter**, the value returned is the absolute value of the counter.

name
A pointer to a character pointer. A successful call will return a pointer to the name of the peer context for which the data value was read.

Return Values

The **RSiGetHotItem, RSiGetHotItemx** subroutine returns a pointer to the current `SpmiHotVals` structure within the hotset. If no more **SpmiHotItems** elements are available, the subroutine returns a NULL value. The structure returned contains the data, such as threshold, which may be relevant for presentation of the results of an **SpmiGetHotSet** subroutine call to end-users. In the returned **SpmiHotVals** structure, all fields contain the correct values as declared, except for the following:

stat
Declared as **SpmiStatHdl**, actually points to a valid `SpmiStat` structure. By casting the handle to a pointer to **SpmiStat**, data in the structure can be accessed.

grandpa
Contains the **cx_handle** for the parent context of the peer contexts.

items
When using the **Spmi** interface this is an array of **SpmiHotItems** structures. When using the **RSiGetHotItem or RSiGetHotItemx** subroutine, the array is empty and attempts to access it will likely result in segmentation faults or access of not valid data.

path
Will contain the path to the parent of the peer contexts. Even when the peer contexts are multiple levels below the parent context, the path points to the top context because the peer context identifiers in the **SpmiHotItems** elements will contain the path name from there and on. For example, if the hotvals peer set defines all volume groups, the path specified in the returned **SpmiHotVals** structure would be "**FS"** and the path name in one **SpmiHotItems** element may be "**rootvg/lv01**". When combined with the metric name from the **stat** field, the full path name can be constructed as, for example, "**FS/rootvg/lv01/%totfree**".

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiGetRawValue or RSiGetRawValuex Subroutine

Purpose

Returns a pointer to a valid SpmiStatVals structure for a given **SpmiStatVals** pointer by extraction from a **data_feed** packet. This subroutine call should only be issued from a callback function after it has been verified that a **data_feed** packet was received from the host identified by the first argument.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiStatVals RSiGetRawValue(rhandle, svp, index)
RSiHandle rhandle;
struct SpmiStatVals *svp;
int *index;
```

```
struct SpmiStatVals RSiGetRawValuex(rhandlex, svp, index)
RSiHandlex rhandlex;
struct SpmiStatVals *svp;
int *index;
```

Description

The **RSiGetRawValue or RSiGetRawValuex** subroutines perform the following actions:

1. Finds an **SpmiStatVals** structure in the received data packet based upon the second argument to the subroutine call. This involves a lookup operation in tables maintained internally by the RSi interface.
2. Updates the **struct SpmiStat** pointer in the **SpmiStatVals** structure to point at a valid **SpmiStat** structure.
3. Returns a pointer to the **SpmiStatVals** structure. The returned pointer points to a static area and is only valid until the next execution of **RSiGetRawValue or RSiGetRawValuex**.
4. Updates an integer variable with the index into the **ValsSet** array of the **data_feed** packet, which corresponds to the second argument to the call.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

svp
A handle of type **struct SpmiStatVals**, which was previously returned by a successful **RSiPathAddSetStat, RSiPathAddSetStatx** subroutine call.

index
A pointer to an integer variable. When the subroutine call succeeds, the index into the **ValsSet** array of the data feed packet is returned. The index corresponds to the element that matches the **svp** argument to the subroutine.

Return Values

If successful, the subroutine returns a pointer; otherwise NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiGetValue or RSiGetValuex Subroutine

Purpose

Returns a data value for a given SpmiStatVals pointer by extraction from the **data_feed** packet. This subroutine call should only be issued from a callback function after it has been verified that a **data_feed** packet was received from the host identified by the first argument.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
float RSiGetValue(rhandle, svp)
RSiHandle rhandle;
struct SpmiStatVals *svp;
```

```
float RSiGetValuex(rhandlex, svp)
RSiHandlex rhandlex;
struct SpmiStatVals *svp;
```

Description

The **RSiGetValue, RSiGetValuex** subroutines provide the following actions:

1. Finds an **SpmiStatVals** structure in the received data packet based upon the second argument to the subroutine call. This involves a lookup operation in tables maintained internally by the RSi interface.

2. Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing based upon its data format.
3. Determines the value as either of type **SiQuantity** or **SiCounter**. If the former is the case, the data value returned is the **val** field in the **SpmiStatVals** structure. If the latter type is found, the value returned by the subroutine is the **val_change** field divided by the elapsed number of seconds since the previous data packet's time stamp.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

svp
A handle of type struct `SpmiStatVals`, which was previously returned by a successful **RSiPathAddSetStat or RSiPathAddSetStatx** subroutine call.

Return Values

If successful, the subroutine returns a non-negative value; otherwise it returns a negative value less than or equal to -1.0. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine's success or failure.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiInit or RSiInitx Subroutine

Purpose

Allocates or changes the table of RSi handles.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
RSiHandle RSiInit(count)
int count;
```

```
RSiHandlex RSiInitx(count)
int count;
```

Description

Before any other **RSi** call is executed, a data-consumer program must issue the **RSiInit or RSiInitx** call and one the following is its purpose :

- Allocate an array of **RSiHandleStruct or RSiHandleStructx** structures and return the address of the array to the data-consumer program.
- Increase the size of a previously allocated array of **RSiHandleStruct or RSiHandleStructx** structures and initialize the new array with the contents of the previous one.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

count
: Must specify the number of elements in the array of RSi handles. If the call is used to expand a previously allocated array, this argument must be larger than the current number of array elements. It must always be larger than zero. Specify the size of the array to be at least as large as the number of hosts your data-consumer program can talk to at any point in time.

Return Values

If successful, the subroutine returns the address of the allocated array. If an error occurs, an error text is placed in the external character array **RSiEMsg** and the subroutine returns NULL. When used to increase the size of a previously allocated array, the subroutine first allocates the new array, then moves the entire old array to the new area. Application programs should, therefore, refer to elements in the RSi handle array by index rather than by address if they anticipate the need for expanding the array. The array only needs to be expanded if the number of remote hosts a data-consumer program talks to might increase over the life of the program.

An application that calls the **RSiInit or RSiInitx** subroutine repeatedly needs to preserve the previous address of the **RSiHandle or RSiHandlex** array while the **RSiInit or RSiInitx** call is re-executed. After the call has completed successfully, the calling program should free the previous array using the **free** subroutine.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiInstantiate or RSiInstantiatex Subroutine

Purpose

Creates (instantiates) all subcontexts of an SpmiCx context object.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiInstantiate(rhandle, context)
RSiHandle rhandle;
cx_handle *context;
```

```
int RSiInstantiatex(rhandlex, context)
RSiHandlex rhandlex;
cx_handle *context;
```

Description

The **RSiInstantiate or RSiInstantiatex** subroutine performs the following actions:

1. Validates that the context identified by the second argument exists.
2. Instantiates the context so that all subcontexts of that context are created in the context hierarchy. Note that this subroutine call currently only makes sense if the context's **SiInstFreq** is set to **SiContInst** or **SiCfgInst** because all other contexts would have been instantiated whenever the **xmservd** daemon was started.

The **RSiInstantiate or RSiInstantiatex** subroutine explicitly instantiates the subcontexts of an instantiable context. If the context is not instantiable, do not call the **RSiInstantiate or RSiInstantiatex**subroutine.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must be an **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

context
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx or RSiPathGetCxx** subroutine call.

Return Values

If successful, the subroutine returns a zero value; otherwise it returns an error code as defined in **SiError** and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];

- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiInvite or RSiInvitex Subroutine

Purpose

Invites data suppliers on the network to identify themselves and returns a table of data-supplier host names.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
char **RSiInvite(resy_callb, excp_callb)
int (*resy_callb)();
int (*excp_callb)();
```

```
char **RSiInvitex(resy_callb, excp_callb)
int (*resy_callb)();
int (*excp_callb)();
```

Description

The **RSiInvite** or **RSiInvitex** subroutine call broadcasts **are_you_there** messages on the network to provoke **xmservd** daemons on remote hosts to respond and returns a table of all responding hosts.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

The arguments to the subroutine are:

resy_callb
Must be either NULL or a pointer to a function that processes the **i_am_back** packets as they are received from the **xmservd** daemons on remote hosts for the duration of the **RSiInvite**, **RSiInvitex** subroutine call. When the callback function is invoked, it is passed three arguments as described in the following information.

If this argument is specified as NULL, a callback function internal to the **RSiInvite**, **RSiInvitex** subroutine receives any **i_am_back** packets and uses them to build the table of host names the function returns.

excp_callb
Must be NULL or a pointer to a function that processes **except_rec** packets as they are received from the **xmservd** daemons on remote hosts. If a NULL pointer is passed, your application does not

receive **except_rec** messages. When this callback function is invoked, it is passed three arguments as described in the following information.

This argument always overrides the corresponding argument of any previous **RSiInvite** or**RSiInvitex**, **RSiOpen** or **RSiOpenx** call, and it can be overridden by subsequent executions of either. In this way, your application can turn exception monitoring on and off. For an **RSiOpen** to override the exception processing specified by a previous open call, the connection must first be closed with the **RSiClose** or **RSiClosex** call. That's because an **RSiOpen** or **RSiOpenx** call against an already active handle is treated as a no-operation.

The **resy_callb** and **excp_callb** functions in your application are called with the following three arguments:

- An **RSiHandle** or **RSiHandlex**. The RSi handle pointed to is almost certain not to represent the host that sent the packet. Ignore this argument, and use only the second one: the pointer to the input buffer.
- A pointer of type **pack *** to the input buffer containing the received packet. Always use this pointer rather than the pointer in the **RSiHandle** or **RSiHandlex** structure.
- A pointer of type **struct sockaddr_in *** or **struct sockaddr_in6 *** to the IP address of the originating host.

Return Values

If successful, the subroutine returns an array of character pointers, each of which contains a host name of a host that responded to the invitation. The returned host names are constructed as two words with the first one being the host name returned by the host in response to an **are_you_there** request; the second one being the character form of the host's IP address. The two words are separated by one or more blanks. This format is suitable as an argument to the **RSiOpen or RSiOpenx** subroutine call. In addition, the external integer variable **RSiInvTabActive** or **RSiInvTabActivex** contains the number of host names found. The returned pointer to an array of host names must not be freed by the subroutine call. The calling program must not assume that the pointer returned by this subroutine call remains valid after subsequent calls to **RSiInvite** or **RSiInvitex**. If the call is not successful, an error text is placed in the external **RSiEMsg** character array, an error number is placed in **RSiErrno**, and the subroutine returns NULL.

The list of host names returned by the **RSiInvite** or **RSiInvitex** does not include the hosts your program has already established a connection with through an **RSiOpen** or **RSiOpenx** call. Your program is responsible for keeping track of such hosts. If you need a list of both sets of hosts, either let the **RSiInvite** or **RSiInvitex** call be the first one issued from your program or merge the list of host names returned by the call with the list of hosts to which you have connections.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiMainLoop or RSiMainLoopx Subroutine

Purpose

Allows an application to suspend execution and wait to get awakened when data feeds arrive.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
void RSiMainLoop(msecs)
int msecs;
```

```
void RSiMainLoopx(msecs)
int msecs;
```

Description

The **RSiMainLoop** or **RSiMainLoopx** subroutine performs the following actions:

1. Allows the data-consumer program to suspend processing while waiting for **data_feed** packets to arrive from one or more **xmservd** daemons.
2. Tells the subroutine that waits for data feeds to return control to the data-consumer program so that the latter can check for and react to other events.
3. Invokes the subroutine to process **data_feed** packets for each such packet received.

To work properly, the **RSiMainLoop** or **RSiMainLoopx** subroutine requires that at least one **RSiOpen or RSiOpenx** call is successfully completed and that the connection is not closed.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

msecs
The minimum elapsed time in milliseconds that the subroutine should continue to attempt receives before returning to the caller. Notice that your program releases control for as many milliseconds you specify but that the callback functions defined on the **RSiOpen** or **RSiOpenx** call may be called repetitively during that time.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiNextCx or RSiNextCxx Subroutine

Purpose

Returns the next subcontext of an SpmiCx context.

Library

RSI Library (**libSpmi**.a)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiCxLink *RSiNextCx(rhandle, context, link, name,
descr)
RSiHandle rhandle;
cx_handle *context;
struct SpmiCxLink *link;
char **name;
char **descr;
```

```
struct SpmiCxLink *RSiNextCxx(rhandlex, context, link, name,
descr)
RSiHandlex rhandlex;
cx_handle *context;
struct SpmiCxLink *link;
char **name;
char **descr;
```

Description

The **RSiNextCx** or **RSiNextCxx** subroutine performs the following actions:

1. Validates that the context identified by the second argument exists.
2. Returns a handle to the next element of the list of subcontexts defined for the context.
3. Returns the short name and description of the subcontext.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

context
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx, RSiPathGetCxx** subroutine call.

link
Must be a pointer to a structure of type **struct SpmiCxLink**, which was previously returned by a successful **RSiFirstCx or RSiFirstCxx** subroutine call or **RSiNextCx** or **RSiNextCxx** subroutine call.

name
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the short name of the subcontext is returned in the character array pointer.

descr
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the description of the subcontext is returned in the character array pointer.

Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiCxLink** . If an error occurs, or if no more subcontexts exist for the context, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiNextStat or RSiNextStatx Subroutine

Purpose

Returns the next statistic of an SpmiCx context.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiStatLink *RSiNextStat (rhandle, context, link, name,
descr)
RSiHandle rhandle;
cx_handle *context;
struct SpmiStatLink *link;
char **name;
char **descr;
```

```
struct SpmiStatLink *RSiNextStatx (rhandlex, context, link, name,
descr)
RSiHandlex rhandlex;
cx_handle *context;
struct SpmiStatLink *link;
char **name;
char **descr;
```

Description

The **RSiNextStat** or **RSiNextStatx** subroutine performs the following actions:

1. Validates that a context identified by the second argument exists.
2. Returns a handle to the next element of the list of statistics defined for the context.
3. Returns the short name and description of the statistic.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

context
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx or RSiPathGetCxx** subroutine call.

link
Must be a pointer to a structure of type **struct SpmiStatLink**, which was previously returned by a successful **RSiFirstStat or RSiFirstStatx** subroutine call or **RSiNextStat or RSiNextStatx** subroutine call.

name
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the short name of the statistics value is returned in the character array pointer.

descr
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the description of the statistics value is returned in the character array pointer.

Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiStatLink**. If an error occurs, or if no more statistics exists for the context, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiOpen or RSiOpenx Subroutine

Purpose

Initializes the RSi interface for a remote host.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiOpen (rhandle, wait, bufsize, hostID, feed_callb,
             resy_callb, excp_callb)
RSiHandle rhandle;
int wait;
int bufsize;
char *hostID;
int (*feed_callb)();
int (*resy_callb)();
int (*excp_callb)();
```

```
int RSiOpenx (rhandlex, wait, bufsize, hostID, feed_callb,
             resy_callb, excp_callb)
RSiHandle rhandlex;
int wait;
int bufsize;
char *hostID;
int (*feed_callb)();
int (*resy_callb)();
int (*excp_callb)();
```

Description

The **RSiOpen** or **RSiOpenx** subroutine performs the following actions:

1. Establishes the issuing data-consumer program as a data consumer known to the **xmservd** daemon on a particular host. The subroutine does this by sending an **are_you_there** packet to the host.
2. Initializes an RSi handle for subsequent use by the data-consumer program.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

The arguments to the subroutine are:

rhandle
Must point to an element of the **RSiHandleStruct** array, which is returned by a previous **RSiInit** call. If the subroutine is successful the structure is initialized and ready to use as a handle for subsequent RSi interface subroutine calls.

rhandlex
Must point to an element of the **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

wait
Must specify the timeout in milliseconds that the RSi interface shall wait for a response when using the request-response functions. On LANs, a reasonable value for this argument is 100 milliseconds. If the response is not received after the specified wait time, the library subroutines retry the receive operation until five times the wait time has elapsed before returning a timeout indication. The wait time must be zero or more milliseconds.

bufsize
Specifies the maximum buffer size to be used for constructing network packets. This size must be at least 4,096 bytes. The buffer size determines the maximum packet length that can be received by your program and sets the limit for the number of data values that can be received in one **data_feed** packet. There's no point in setting the buffer size larger than that of the **xmservd** daemon because both must be able to handle the packets. If you need large sets of values, you can use the command line argument **-b** of **xmservd** to increase its buffer size up to 16,384 bytes.

The fixed part of a **data_feed** packet is 104 bytes and each value takes 32 bytes. A buffer size of 4,096 bytes allows up to 124 values per packet.

hostID
Must be a character array containing the identification of the remote host whose **xmservd** daemon is the one with which you want to talk. The first characters of the host identification (up to the first white space) is used as the host name. The full host identification is stored in the **RSiHandle** field **longname** and may contain any description that helps the user to identify the host used. The host name may be either in long format (including domain name) or in short format.

feed_callb
Must be a pointer to a function that processes **data_feed** packets as they are received from the **xmservd** daemon. When this callback function is invoked, it is passed three arguments as described in the following information.

resy_callb
Must be a pointer to a function that processes **i_am_back** packets as they are received from the **xmservd** daemon. When this callback function is invoked it is passed three arguments as described in the following information.

excp_callb
Must be NULL or a pointer to a function that processes the **except_rec** packets as they are received from the **xmservd** daemon. If a NULL pointer is passed, your application does not receive **except_rec** messages. When this callback function is invoked, it is passed three arguments as described in the following information. This argument always overrides the corresponding argument of any previous **RSiInvite or RSiInvitex** subroutine or **RSiOpen or RSiOpenx** subroutine call and can itself be overridden by subsequent executions of either. In this way, your application can turn exception monitoring on and off. For an **RSiOpen or RSiOpenx** call to override the exception processing specified by a previous open call, the connection must first be closed with the **RSiClose or RSiClosex** subroutine call.

The **feed_callb**, **resy_callb**, and **excp_callb** functions are called with the following arguments:

- **RSiHandle or RSiHandlex** – When a **data_feed** packet is received, the structure pointed to is guaranteed to represent the host sending the packet. In all other situations the **RSiHandle or RSiHandlex** structure may represent any of the hosts to which your application is communicating.

Pointer of type **pack *** to the input buffer containing the received packet. In callback functions, always use this pointer rather than the pointer in the **RSiHandle or RSiHandlex** structure.

Pointer of type **struct sockaddr_in *** or **struct sockadd_in 6*** to the IP address of the originating host.

Return Values

If successful, the subroutine returns zero and initializes the array element of type **RSiHandle or RSiHandlex** pointed to by **rhandle or rhandlex**. If an error occurs, error text is placed in the external character array **RSiEMsg** and the subroutine returns a negative value.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiPathAddSetStat or RSiPathAddSetStatx Subroutine

Purpose

Add a single statistics value to an already defined SpmiStatSet.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
struct SpmiStatVals *RSiPathAddSetStat (rhandle, statset,
path)
RSiHandle rhandle;
struct SpmiStatSet *statset;
char *path;
```

```
struct SpmiStatVals *RSiPathAddSetStatx (rhandlex, statset,
path)
RSiHandle rhandlex;
struct SpmiStatSet *statset;
char *path;
```

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandle
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

statset
Must be a pointer to a structure of type **struct SpmiStatSet**, which was previously returned by a successful **RSiCreateStatSet or RSiCreateStatSetx** subroutine call.

path
Must be the full value path name of the statistics value to add to the **SpmiStatSet**. The value path name must not include a terminating slash. Note that value path names never start with a slash.

Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiStatVals**. If an error occurs, NULL is returned and an error text may be placed in the external character array **RSiEMsg**. If you attempt to add more values to a statset than the current local buffer size allows, **RSiErrno** is set to **RSiTooMany**. If you attempt to add more values than the buffer size of the remote host's **xmservd** daemon allows, **RSiErrno** is set to **RSiBadStat** and the status field in the returned packet is set to **too_many_values**.

The external integer **RSiMaxValues** holds the maximum number of values acceptable with the data-consumer's buffer size.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiPathGetCx or RSiPathGetCxx Subroutine

Purpose

Searches the context hierarchy for an SpmiCx context that matches a context path name.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
cx_handle *RSiPathGetCx (rhandle, path)
RSiHandle rhandle;
char *path;
```

```
cx_handle *RSiPathGetCxx (rhandlex, path)
RSiHandlex  rhandlex;
char *path;
```

Description

The **RSiPathGetCx or RSiPathGetCxx** subroutine performs the following actions:

1. Searches the context hierarchy for a given path name of a context.
2. Returns a handle to be used when subsequently referencing the context.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

path
A path name of a context for which a handle is to be returned. The context path name must be the full path name and must not include a terminating slash. Note that context path names never start with a slash.

Return Values

If successful, the subroutine returns a handle defined as a pointer to a structure of type **cx_handle**. If an error occurs, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiStartFeed or RSiStartFeedx Subroutine

Purpose

Tells **xmservd** to start sending data feeds for a statset.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiStartFeed (rhandle, statset, msecs)
RSiHandle rhandle;
struct SpmiStatSet *statset;
int msecs;
```

```
int RSiStartFeedx (rhandlex, statset, msecs)
RSiHandlex rhandlex;
struct SpmiStatSet *statset;
int msecs;
```

Description

The **RSiStartFeed or RSiStartFeedx** subroutine performs the following function:

1. Informs **xmservd** of the frequency with which it is required to send **data_feed** packets.
2. Tells the **xmservd** to start sending **data_feed** packets.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

statset
Must be a pointer to a structure of type **struct SpmiStatSet** , which was previously returned by a successful **RSiCreateStatSet or RSiCreateStatSetx** subroutine call.

msecs
The number of milliseconds between the sending of **data_feed** packets. This number is rounded to a multiple of **min_remote_int** milliseconds by the **xmservd** daemon on the remote host. This minimum interval can be modified through the **-i** command line interval to **xmservd**.

Return Values

If successful, the subroutine returns zero; otherwise it returns -1 and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiStartHotFeed or RSiStartHotFeedx Subroutine

Purpose

Tells **xmservd** to start sending hot feeds for a hotset or to start checking for if exceptions or SNMP traps should be generated.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiStartFeed (rhandle, hotset, msecs)
RSiHandle rhandle;
struct SpmiHotSet *hotset;
int msecs;
```

```
int RSiStartFeedx (rhandlex, hotset, msecs)
RSiHandlex rhandlex;
struct SpmiHotSet *hotset;
int msecs;
```

Description

The **RSiStartHotFeed or RSiStartHotFeedx** subroutine performs the following function:

1. Informs **xmservd** of the frequency with which it is required to send **hot_feed** packets, if the hotset is defined to generate **hot_feed** packets.
2. Informs **xmservd** of the frequency with which it is required to check if exceptions or SNMP traps should be generated. This is only done if it is specified for the hotset that exceptions and/or SNMP traps should be generated.
3. Tells the **xmservd** to start sending **data_feed** packets and/or start checking for exceptions or traps.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandle
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

hotset
Must be a pointer to a structure of type **struct SpmiHotSet**, which was previously returned by a successful **RSiCreateHot or RSiCreateHotx** subroutine call.

msecs
The number of milliseconds between the sending of **hot_feed** packets and/or the number of milliseconds between checks for if exceptions or SNMP traps should be generated. This number is rounded to a multiple of **min_remote_int** milliseconds by the **xmservd** daemon on the remote host. This minimum interval can be modified through the **-i** command line interval to **xmservd**.

Return Values

If successful, the subroutine returns zero; otherwise it returns -1 and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiStatGetPath or RSiStatGetPathx Subroutine

This subroutine is part of the Performance Toolbox for AIX licensed product.

Purpose

Finds the full path name of a statistic identified by a `SpmiStatVals` pointer.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
char *RSiStatGetPath (rhandle, svp)
RSiHandle rhandle;
struct SpmiStatVals *svp;
```

```
char *RSiStatGetPathx (rhandlex, svp)
RSiHandlex rhandlex;
struct SpmiStatVals *svp;
```

Description

The **RSiStatGetPath or RSiStatGetPathx** subroutine performs the following actions:

1. Validates that the **SpmiStatVals** statistic identified by the second argument does exist.
2. Returns a pointer to a character array containing the full value path name of the statistic.

The memory area pointed to by the returned pointer is freed when the **RSiStatGetPath or RSiStatGetPathx** subroutine call is repeated. For each invocation of the subroutine, a new memory area is allocated and its address is returned.

If the calling program needs the returned character string after issuing the **RSiStatGetPath or RSiStatGetPathx** subroutine call, the program must copy the returned string to locally allocated memory before reissuing the subroutine call.

Parameters

rhandle
Must point to an **RSiHandle** handle which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must point to an **RSiHandlex** handle which was previously initialized by the **RSiOpenx** subroutine.

svp
Must be a handle of type **struct SpmiStatVals** as returned by a successful **RSiPathAddSetStat or RSiPathAddSetStatx** subroutine call.

Return Values

If successful, the **RSiStatGetPath or RSiStatGetPathx** subroutine returns a pointer to a character array containing the full path name of the statistic. If unsuccessful, the subroutine returns a NULL value and an error text may be placed in the external character array **RSiEMsg**.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiStopFeed or RSiStopFeedx Subroutine

Purpose

Tells **xmservd** to stop sending data feeds for a statset.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiStopFeed(rhandle, statset, erase)
RSiHandle rhandle;
```

```
struct SpmiStatSet *statset;
boolean erase;
```

```
int RSiStopFeedx (rhandlex, statset, erase)
RSiHandlex rhandlex;
struct SpmiStatSet *statset;
boolean erase;
```

Description

The **RSiStopFeed or RSiStopFeedx** subroutine instructs the **xmservd** of a remote system to:

1. Stop sending **data_feed** packets for a given SpmiStatSet. If the daemon is not told to erase the **SpmiStatSet**, feeding of data can be resumed by issuing the **RSiStartFeed or RSiStartFeedx** subroutine call for the **SpmiStatSet**.
2. Optionally tells the daemon and the API library subroutines to erase all their information about the **SpmiStatSet**. Subsequent references to the erased **SpmiStatSet** are not valid.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

statset
Must be a pointer to a structure of type **struct SpmiStatSet**, which was previously returned by a successful **RSiCreateStatSet or RSiCreateStatSetx** subroutine call. Data feeding must have started for this **SpmiStatSet** via a previous **RSiStartFeed or RSiStartFeedx** subroutine call.

erase
If this argument is set to true, the **xmservd** daemon on the remote host discards all information about the named **SpmiStatSet**. Otherwise the daemon maintains its definition of the set of statistics.

Return Values

If successful, the subroutine returns zero, otherwise -1. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine's success or failure.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSi**Errno variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

RSiStopHotFeed or RSiStopHotFeedx Subroutine

Purpose

Tells **xmservd** to stop sending hot feeds for a hotset and to stop checking for exception and SNMP trap generation.

Library

RSI Library (**libSpmi.a**)

Syntax

```
#include sys/Rsi.h
```

```
int RSiStopFeed (rhandle, hotset, erase)
RSiHandle rhandle;
struct SpmiHotSet *hotset;
boolean erase;
```

```
int RSiStopFeedx (rhandlex, hotset, erase)
RSiHandlex rhandlex;
struct SpmiHotSet *hotset;
boolean erase;
```

Description

The **RSiStopHotFeed or RSiStopHotFeedx** subroutine instructs the **xmservd** of a remote system to:

1. Stop sending **hot_feed** packets or check if exceptions or SNMP traps should be generated for a given SpmiHotSet. If the daemon is not told to erase the **SpmiHotSet**, feeding of data can be resumed by issuing the **RSiStartHotFeed or RSiStartHotFeedx** subroutine call for the **SpmiHotSet**.
2. Optionally tells the daemon and the API library subroutines to erase all their information about the **SpmiHotSet**. Subsequent references to the erased **SpmiHotSet** are not valid.

This subroutine is part of the Performance Toolbox for AIX licensed product.

Parameters

rhandle
Must point to a valid **RSiHandle** handle, which was previously initialized by the **RSiOpen** subroutine.

rhandlex
Must point to a valid **RSiHandlex** handle, which was previously initialized by the **RSiOpenx** subroutine.

hotset
Must be a pointer to a structure of type **struct SpmiHotSet**, which was previously returned by a successful **RSiCreateHotSet or RSiCreateHotSetx** subroutine call. Data feeding must have been started for this **SpmiStatSet** via a previous **RSiStartHotFeed or RSiStartHotFeedx** subroutine call.

erase
If this argument is set to true, the **xmservd** daemon on the remote host discards all information about the named **SpmiHotSet**. Otherwise the daemon maintains its definition of the set of statistics.

Return Values

If successful, the subroutine returns zero, otherwise -1. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine's success or failure.

Error Codes

All Remote Statistic Interface (RSI) subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSi**Errno variable returns an error code, as defined in the enum **RSiErrorType**.

Files

Item	Description
/usr/include/sys/Rsi.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

rs_alloc Subroutine

Purpose

Allocates a resource set and returns its handle.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
rsethandle_t rs_alloc (flags)
unsigned int flags;
```

Description

The **rs_alloc** subroutine allocates a resource set and initializes it according to the information specified by the *flags* parameter. The value of the *flags* parameter determines how the new resource set is initialized.

The handle for the new resource set is returned by the subroutine.

Parameters

Item	Description
flags	Specifies how the new resource set is initialized. It takes one of the following values, defined in **rset.h**: • **RS_EMPTY** (or 0 value): The resource set is initialized to contain no resources. • **RS_SYSTEM**: The resource set is initialized to contain available system resources. • **RS_ALL**: The resource set is initialized to contain all resources. • **RS_PARTITION**: The resource set is initialized to contain the resources in the caller's process partition resource set.

Return Values

On successful completion, a resource set handle for the new resource set is returned. Otherwise, a value of 0 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_alloc** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	The *flags* parameter contains an invalid value.
ENOMEM	There is not enough space to create the data structures related to the resource set.

rs_discardname Subroutine

Purpose

Discards a resource set definition from the system resource set registry.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_discardname(namespace, rsname)
char *namespace, *rsname;
```

Description

The **rs_discardname** subroutine discards from the system global repository the definition of the resource set. The resource set is identified by the *namespace* and *rsname* parameters. The specified resource set is removed from the registry, and can no longer be shared with other applications.

In order to be able to discard a name from the global repository, the calling process must have root authority or CAP_NUMA_ATTACH capability, and an effective user ID equal to that of the *rsname* parameter's creator. CAP_NUMA_ATTACH allows non-root users to create or remove an exclusive *rset*.

The `rs_discardname` subroutine is used to remove an exclusive *rset*. When an exclusive *rset* is removed, the state of CPUs in that *rset* is modified so that those CPUs can run any work on the system. Root authority is required to remove an exclusive *rset*. See Exclusive use processor resource sets in *Operating system and device management* and the rmrset command for more information.

Parameters

Item	Description
namespace	Points to a null terminated string corresponding to the name space within which *rsname* should be found.
rsname	Points to a null terminated string corresponding to the name of a registered resource set to be discarded.

Return Values

If successful, a value of 0 is returned. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_discardname** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *rsname* parameter contains a null value. • The *namespace* parameter contains a null value. • The *rsname* or *namespace* parameters point to an invalid name. • The name length is null or greater than the RSET_NAME_SIZE constant (defined in **rset.h**), or the name contains invalid characters.
EPERM	One of the following is true: • The calling process has neither root authority nor CAP_NUMA_ATTACH capability. • The calling process has neither the same user ID as the creator of the *rsname* definition nor root authority . • The *namespace* parameter starts with sys. This name space is reserved for system use.
EFAULT	Invalid address, and/or exceptions outside **errno** range.

rs_free Subroutine

Purpose

Frees a resource set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
void rs_free(rset)
rsethandle_t rset;
```

Description

The **rs_free** subroutine frees a resource set identified by the *rset* parameter. The resource set must have been allocated by the **rs_alloc** subroutine

Parameters

Item	Description
rset	Specifies the resource set whose memory will be freed.

rs_getassociativity Subroutine

Purpose

Gets the hardware associativity values for a resource.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_getassociativity (type, id, assoc_array, array_size)
unsigned int type;
unsigned int id;
unsigned int *assoc_array;
unsigned int array_size;
```

Description

The **rs_getassociativity** subroutine returns the array of hardware associativity values for a specified resource.

This is a special purpose subroutine intended for specialized root applications needing the hardware associativity value information. The **rs_getinfo**, **rs_getrad**, and **rs_numrads** subroutines are provided for non-root applications to discover system hardware topology.

The calling process must have root authority to get hardware associativity values.

Parameters

Item	Description
type	Specifies the resource type whose associativity values are requested. The only value supported to retrieve values for a processor is R_PROCS.
id	Specifies the logical resource id whose associativity values are requested.
assoc_array	Specifies the address of an array of unsigned integers to receive the associativity values.
array_size	Specifies the number of unsigned integers in *assoc_array*.

Return Values

If successful, a value of 0 is returned. The *assoc_array* parameter array contains the resource's associativity values. The first entry in the array indicates the number of associativity values returned. If the hardware system does not provide system topology data, a value of 0 is returned in the first array entry. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_getassociativity** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following occurred: • The *array_size* parameter was specified as 0. • An invalid *type* parameter was specified.
ENODEV	The resource specified by the *id* parameter does not exist.
EFAULT	Invalid address.
EPERM	The calling process does not have root authority.

rs_get_homesrad Subroutine

Purpose

Gets the currently running thread's home SRADID (Scheduler Resource Allocation Domain Identifier).

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
sradid_t rs_get_homesrad(void)
```

Description

If the ENHANCED_AFFINITY services are enabled, the **rs_get_homesrad** subroutine returns the home SRADID of the currently running thread. If the ENHANCED_AFFINITY services are not enabled, the **rs_get_homesrad** subroutine returns SRADID_ANY. SRADID is the index of a resource allocation domain (RAD) at the R_SRADSDL system detail level. See the “rs_getrad Subroutine ” on page 1807 subroutine for information about obtaining a resource set that corresponds to a returned SRADID.

Return Values

If the ENHANCED_AFFINITY services are enabled, the home SRADID of the currently running thread is returned. Otherwise, SRADID_ANY is returned.

rs_getinfo Subroutine

Purpose

Gets information about a resource set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_getinfo(rset, info_type, flags)
rsethandle_t rset;
rsinfo_t info_type;
unsigned int flags;
```

Description

The **rs_getinfo** subroutine retrieves information about the resource set identified by the *rset* parameter. Depending on the value of the *info_type* parameter, the **rs_getinfo** subroutine returns information about the number of available processors, the number of available memory pools, or the amount of available memory contained in the resource *rset*. The subroutine can also return global system information such as the maximum system detail level, the symmetric multiprocessor (SMP) and multiple chip module (MCM) system detail levels, and the maximum number of processor or memory pool resources in a resource set.

Parameters

Item	Description
rset	Specifies a resource set handle of a resource set the information should be retrieved from. This parameter is not meaningful if the *info_type* parameter is R_MAXSDL, R_MAXPROCS, R_MAXMEMPS, R_SMPSDL, or R_MCMSDL.

Item	Description
info_type	Specifies the type of information being requested. One of the following values (defined in **rset.h**) can be used: • **R_LGPGDEF**: The number of defined large pages in the resource set is returned in units of megabytes. • **R_LGPGFREE**: The number of free large pages in the resource set is returned in units of megabytes. • **R_NUMPROCS**: The number of available processors in the resource set is returned. • **R_NUMMEMPS**: The number of available memory pools in the resource set is returned. • **R_MEMSIZE**: The amount of available memory (in MB) contained in the resource set is returned. • **R_MAXSDL**: The maximum system detail level of the system is returned. • **R_MAXPROCS**: The maximum number of processors that may be contained in a resource set is returned. • **R_MAXMEMPS**: The maximum number of memory pools that may be contained in a resource set is returned. • **R_SMPSDL**: The system detail level that corresponds to the traditional notion of an SMP is returned. A system detail level of 0 is returned if the hardware system does not provide system topology data. • **R_MCMSDL**: The system detail level that corresponds to resources packaged in an MCM is returned. A system detail level of 0 is returned if the hardware system does not have MCMs or does not provide system topology data. • **R_SRADSDL**: The system detail level that corresponds to system's scheduler resource allocation domain is returned. This SDL is the basis for most affinity resource allocation and scheduling activities. This SDL identifies resources that have a local relationship. • **R_REF1SDL**: The system detail level of the first hardware provided affinity reference point. This SDL identifies resources that have a near relationship. Only some hardware systems provide a R_REF1SDL reference point. On systems that do not provide a reference point, the R_REF1SDL will identify the R_SRADSDL system detail level. • **R_MAXSRADS**: The maximum number of RADs at the R_SRADSDL system detail level is returned. • **R_GENERATION**: The generation number of the system's current resource set topology is returned. The number increases whenever a change to the system's resource set topology occurs. For example, the dynamic reconfiguration that adds a CPU to the system causes the generation number to increase.
flags	Reserved for future use. Specify as 0.

Return Values

If successful, the requested information is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_getinfo** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *info_type* parameter specifies an invalid resource type value. • The *flags* parameter was not specified as 0.
EFAULT	Invalid address.

rs_getnameattr Subroutine

Purpose

Retrieves the access control information of a resource set definition in the system resource set registry.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_getnameattr(namespace, rsname, attr)
char *namespace, *rsname;
rs_attributes_t *attr;
```

Description

The **rs_getnameattr** subroutine retrieves from the system resource set registry the access control information of the resource set definition specified by the *namespace* and *rsname* parameters.

The owner ID, group ID, and access control information of the specified resource set are stored in the structure pointed to by the *attr* parameter.

Note: No special authority or access permission is required to query this information.

Parameters

Item	Description
namespace	Points to a null terminated string corresponding to the name space within which the *rsname* parameter should be found.
rsname	Points to a null terminated string corresponding to the name the information should be retrieved for.

Item	Description
attr	Points to an **rs_attributes_t** structure containing the *owner*, *group*, and *mode* fields, which will be filled by the subroutine. The *mode* field in the **rs_attributes_t** structure is used to store the access permissions, and is constructed by logically ORing one or more of the following values, defined in **rset.h**: • **RS_IRUSR**: Gives read rights to the name's owner. • **RS_IWUSR**: Gives write rights to the name's owner. • **RS_IRGRP**: Gives read rights to users of the same group as the name's owner. • **RS_IWGRP**: Gives write rights to users of the same group as the name's owner. • **RS_IROTH**: Gives read rights to others. • **RS_IWOTH**: Gives write rights to others. Read privilege for a user means that the user can retrieve a resource set definition by issuing a call to the **rs_getnamedrset** subroutine. Write privilege for a user means that the user can redefine a name by issuing another call to the **rs_getnamedrset** subroutine.

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_getnameattr** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	If one of the following is true: • The *rsname* parameter is a null pointer. • The *namespace* parameter is a null pointer. • The *rsname* or *namespace* parameters point to an invalid name. The name length is 0 or greater than the RSET_NAME_SIZE constant (defined in **rset.h**), or the *rsname* parameter contains invalid characters.
ENOENT	The *rsname* parameter could not be found in the name space identified by the *namespace* parameter.
EFAULT	Invalid address.

rs_getnamedrset Subroutine

Purpose

Retrieves the contents of a named resource set from the system resource set registry.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_getnamedrset (namespace, rsname, rset)
char *namespace, *rsname;
```

Description

The **rs_getnamedrset** subroutine retrieves a resource set definition from the system registry. The *namespace* and *rsname* parameters identify the resource set to be retrieved. The *rset* parameter identifies where the retrieved resource set should be returned. The *namespace* and *rsname* parameters identify a previously registered resource set definition.

The calling process must have root authority or read access rights to the resource set definition in order to retrieve it.

The *rset* parameter must be allocated (using the **rs_alloc** subroutine) prior to calling the **rs_getnamedrset** subroutine.

Parameters

Item	Description
namespace	Points to a null-terminated string corresponding to the name space within which *rsname* is found.
rsname	Points to a null-terminated string corresponding to the previously registered name of a resource set.
rset	Specifies the resource set handle for the resource set that the registered resource set is copied into. The registered resource set is specified by the *rsname* parameter.

Return Values

If successful , a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_getnamedrset** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *rsname* parameter is a null pointer. • The *namespace* parameter is a null pointer. • The *rsname* or *namespace* parameters point to an invalid name. The name length is 0 or greater than the RSET_NAME_SIZE constant (defined in **rset.h**), or the *rsname* parameter contains invalid characters.
ENOENT	The *rsname* parameter could not be found in the name space identified by the *namespace* parameter.
EPERM	The calling process has neither read permission on *rsname* nor root authority.
EFAULT	Invalid address.

rs_getpartition Subroutine

Purpose

Gets the partition resource set to which a process is attached.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_getpartition (pid, rset)
pid_t pid;
rsethandle_t rset;
```

Description

The **rs_getpartition** subroutine returns the partition resource set attached to the specified process. A process ID value of RS_MYSELF indicates the partition resource set attached to the current process is requested.

The return value from the **rs_getpartition** subroutine indicates the type of resource set returned.

A value of RS_PARTITION_RSET indicates the process has a partition resource set that is set explicitly. This may be set with the **rs_setpartition** subroutine or through the use of WLM work classes with resource sets.

A value of RS_DEFAULT_RSET indicates the process did not have an explicitly set partition resource set. The system default resource set is returned.

Parameters

Item	Description
pid	Specifies the process ID whose partition *rset* is requested.
rset	Specifies the resource set to receive the process' partition resource set.

Return Values

If successful, a value of RS_PARTITION_RSET, or RS_DEFAULT_RSET is returned. If unsuccessful, a value of -1 is returned and the global **errno** variable is set to indicate the error.

Error Codes

The **rs_getpartition** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EFAULT	Invalid address.
ESRCH	The process identified by the *pid* parameter does not exist.

rs_getrad Subroutine

Purpose

Returns a system resource allocation domain (RAD) contained in an input resource set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_getrad (rset, rad, sdl, index, flags)
rsethandle_t rset, rad;
unsigned int sdl;
```

```
unsigned int index;
unsigned int flags;
```

Description

The **rs_getrad** subroutine returns a system RAD at a specified system detail level and index that is contained in an input resource set. If only some of the resources in the specified system RAD are contained in the input resource set, only the resources in both the system RAD and the input resource set are returned.

The input resource set is specified by the *rset* parameter. The output system RAD is identified by the *rad* parameter.

The system RAD is specified by system detail level *sdl* and index number *index*. If only a portion of the specified RAD is contained in *rset*, only that portion is returned in *rad*.

The *rset* and *rad* parameters must be allocated (using the **rs_alloc** subroutine) prior to calling the **rs_getrad** subroutine.

Parameters

Item	Description
rset	Specifies a resource set handle for the input resource set.
rad	Specifies a resource set handle to receive the desired system RAD (contained in the *rset* parameter).
sdl	Specifies the system detail level of the desired system RAD.
index	Specifies the index of the system RAD that should be returned from among those at the specified *sdl*. This parameter must belong to the **[0, rs_numrads(***rset, sdl, flags***)- 1]** interval.
flags	The following flags (defined in **rset.h**) can be used to modify the default behavior of the **rs_getrad** subroutine. By default, the **rs_getrad** subroutine empties the resource set specified by *rad* before the specified RAD is retrieved. • **RS_UNION**: Instead of emptying *rad* before the specified RAD is retrieved, the RAD retrieved is added to the contents of *rad*. On completion, *rad* contains the union of its original contents and the specified RAD. • **RS_EXCLUSION**: Instead of emptying *rad* before the specified RAD is retrieved, the resources in the specified RAD that are also in *rad* are removed from *rad*. On return, *rad* contains all the resources it originally contained except those in the specified RAD.

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_getrad** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *flags* parameter contains an invalid value. • The *sdl* parameter is greater than the maximum system detail level. • The RAD specified by the *index* parameter does not exist at the system detail level specified by the *sdl* parameter.
EFAULT	Invalid address.

rs_info Subroutine

Purpose

Retrieves system affinity information.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
long rs_info(void *out, long command, long arg1, long arg2)
```

Description

The **rs_info** subroutine returns affinity system information.

Parameters

Item	Description
out	Specifies the address where the affinity request information is optionally entered and where output information is returned.
command	Specifies the requested affinity information. The command parameter has the following values:

RS_CONTAINING_RAD
Returns the index number of the resource allocation domain at the previous (next lower number) system detail level that contains the resource allocation domain specified by the *arg1* and *arg2* parameters. The *arg1* parameter specifies the system detail level number of requested resource allocation domain. The *arg2* parameter specifies the index of the resource allocation domain within the *arg1* system detail level. The **out* parameter points to an unsigned integer that receives the containing resource allocation domain index.

RS_SRADID_LOADAVG
Returns the dispatcher load average for the available CPUs in a specified SRADID (Scheduler Resource Allocation Domain Identifier). The *arg1* parameter specifies the SRADID whose load average is requested. The *arg2* parameter specifies the size of the output parameter area provided in the out parameter. The *out* parameter points to the address of a loadavg_info_t structure to receive the output of the query. The **rs_info()** subroutine returns the load average and the number of available CPUs in the SRADID in the loadavg_info_t structure.

RS_SRADID_USABLE_LOADAVG
Returns the dispatcher load average for the available CPUs in a specified SRADID that can be used by the calling thread. The *arg1* parameter specifies the SRADID whose load average is requested. CPUs in the specified SRADID that the calling thread cannot use due to process or thread resource set attachments or system exclusive resource sets are excluded from the load average calculation. The *arg2* parameter specifies the size of the output parameter area provided in the *out* parameter. The *out* parameter points to the address of a loadavg_info_t structure to receive the output of the query. The **rs_info()** subroutine returns the load average and number of usable CPUs in the SRADID in the loadavg_info_t structure.

arg1	Specifies the parameter information that depends on the *command* parameter.
arg2	Specifies the parameter information that depends on the *command* parameter.

Return Values

If successful, the requested information is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EFAULT	The read or write of the **out* parameter is not successful.
EINVAL	One of the following occurred: • An invalid *command* argument is specified. • An invalid *arg1* or *arg2* parameter is specified.

rs_init Subroutine

Purpose

Initializes a previously allocated resource set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_init (rset, flags)
rsethandle_t rset;
unsigned int flags;
```

Description

The **rs_init** subroutine initializes a previously allocated resource set. The resource set is initialized according to information specified by the *flags* parameter.

Parameters

Item	Description
rset	Specifies the handle of the resource set to initialize.
flags	Specifies how the resource set is initialized. It takes one of the following values, defined in **rset.h**: • **RS_EMPTY**: The resource set is initialized to contain no resources. • **RS_SYSTEM**: The resource set is initialized to contain available system resources. • **RS_ALL**: The resource set is initialized to contain all resources. • **RS_PARTITION**: The resource set is initialized to contain the resources in the caller's process partition resource set.

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_init** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	The *flags* parameter contains an invalid value.

rs_numrads Subroutine

Purpose

Returns the number of system resource allocation domains (RADs) that have available resources.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_numrads(rset, sdl, flags)
rsethandle_t rset;
unsigned int sdl;
unsigned int flags;
```

Description

The **rs_numrads** subroutine returns the number of system RADs at system detail level *sdl*, that have available resources contained in the resource set identified by the *rset* parameter.

The number of atomic RADs contained in the *rset* parameter is returned if the *sdl* parameter is equal to the maximum system detail level.

Parameters

Item	Description
rset	Specifies the resource set handle for the resource set being queried.
sdl	Specifies the system detail level in which the caller is interested.
flags	Reserved for future use. Specify as 0.

Return Values

If successful, the number of available RADs at system detail level *sdl*, that have resources contained in the specified resource set is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_numrads** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • The *flags* parameter contains an invalid value. • The *sdl* parameter is greater than the maximum system detail level.

Item	Description
EFAULT	Invalid address.

rs_op Subroutine

Purpose

Performs a set of operations on one or two resource sets.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_op (command, rset1, rset2, flags, id)
unsigned int command;
rsethandle_t rset1, rset2;
unsigned int flags;
unsigned int id;
```

Description

The **rs_op** subroutine performs the operation specified by the *command* parameter on resource set *rset1* or both resource sets *rset1* and *rset2*.

Parameters

Item	Description
command	Specifies the operation to apply to the resource sets identified by *rset1* and *rset2*. One of the following values, defined in **rset.h**, can be used: • **RS_UNION**: The resources contained in either *rset1* or *rset2* are stored in *rset2*. • **RS_INTERSECTION**: The resources that are contained in both *rset1* and *rset2* are stored in *rset2*. • **RS_EXCLUSION**: The resources in *rset1* that are also in *rset2* are removed from *rset2*. On completion, *rset2* contains all the resources that were contained in *rset2* but were not contained in *rset1*. • **RS_COPY**: All resources in *rset1* whose type is *flags* are stored in *rset2*. If *rset1* contains no resources of this type, *rset2* will be empty. The previous content of *rset2* is lost, while the content of *rset1* is unchanged. • **RS_FIRST**: The first resource whose type is *flags* is retrieved from *rset1* and stored in *rset2*. If *rset1* contains no resources of this type, *rset2* will be empty. • **RS_NEXT**: The resource from *rset1* whose type is *flags* and that follows the resource contained in *rset2* is retrieved and stored in *rset2*. If no resource of the appropriate type follows the resource specified in *rset2*, *rset2* will be empty. • **RS_NEXT_WRAP**: The resource from *rset1* whose type is *flags* and that follows the resource contained in *rset2* is retrieved and stored in *rset2*. If no resource of the appropriate type follows the resource specified in *rset2*, *rset2* will contain the first resource of this type in *rset1*. • **RS_ISEMPTY**: Test if resource set *rset1* is empty. • **RS_ISEQUAL**: Test if resource sets *rset1* and *rset2* are equal. • **RS_ISCONTAINED**: Test if all resources in resource set *rset1* are also contained in resource set *rset2*. • **RS_TESTRESOURCE**: Test if the resource whose type is *flags* and index is *id* is contained in resource set *rset1*. • **RS_ADDRESOURCE**: Add the resource whose type is *flags* and index is *id* to resource set *rset1*. • **RS_DELRESOURCE**: Delete the resource whose type is *flags* and index is *id* from resource set *rset1*. • RS_STSET: Constructs an ST resource set by including only one hardware thread per physical processor included in *rset1* and stores it in *rset2*. Only available processors are considered when constructing the ST resource set.
rset1	Specifies the resource set handle for the first of the resource sets involved in the *command* operation.
rset2	Specifies the resource set handle for the second of the resource sets involved in the *command* operation. This resource set is also used, on return, to store the result of the operation, and its previous content is lost. The *rset2* parameter is ignored on the RS_ISEMPTY, RS_TESTRESOURCE, RS_ADDRESOURCE, and RS_DELRESOURCE commands.

Item	Description
flags	When combined with the RS_COPY command, the *flags* parameter specifies the type of the resources that will be copied from *rset1* to *rset2*. When combined with an RS_FIRST or an RS_NEXT command, the *flags* parameter specifies the type of the resource that will be retrieved from *rset1*. This parameter is constructed by logically ORing one or more of the following values, defined in **rset.h**: • **R_PROCS:** processors • **R_MEMPS:** memory pools • **R_ALL_RESOURCES:** processors and memory pools If none of the above are specified for *flags*, R_ALL_RESOURCES is assumed.
id	On the RS_TESTRESOURCE, RS_ADDRESOURCE, and RS_DELRESOURCE commands, the *id* parameter specifies the index of the resource to be tested, added, or deleted. This parameter is ignored on the other commands.

Return Values

If successful, the commands RS_ISEMPTY, RS_ISEQUAL, RS_ISCONTAINED, and RS_TESTRESOURCE return 0 if the tested condition is not met and 1 if the tested condition is met. All other commands return 0 if successful. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_op** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	If one of the following is true: • *rset1* identifies an invalid resource set. • *rset2* identifies an invalid resource set. • *command* identifies an invalid operation. • *command* is RS_NEXT or RS_NEXT_WRAP*, and *rset2* does not contain a single resource. • *command* is RS_NEXT or RS_NEXT_WRAP*, and the single resource contained in *rset2* is not also contained in *rset1*. • *flags* identifies an invalid resource type. • *id* specifies a resource index that is too large.
EFAULT	Invalid address.

rs_registername Subroutine

Purpose

Registers a resource set definition in the system resource set registry.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_registername(rset, namespace, rsname, mode, command)
rsethandle_t rset;
char *namespace, *rsname;
unsigned int mode, command;
```

Description

The **rs_registername** subroutine registers in the system resource registry (within the name space identified by *namespace*) the definition of the resource set identified by the *rset* handle. The **rs_registername** subroutine does this by associating with it the name specified by the null terminated string structure pointed to by *rsname*.

If *rsname* does not exist, the owner and group IDs of *rsname* are set to the caller's owner and group IDs, and the access control information for *rsname* is set according to the *mode* parameter.

If *rsname* already exists, its owner and group IDs and its access control information are left unchanged, and the *mode* parameter is ignored. This name can be shared with any applications to identify a dedicated resource set.

Using the *command* parameter, you can ask to overwrite or not to overwrite the *rsname* parameter's registration if it already exists in the global repository within the name space identified by *namespace*. If *rsname* already exists within the specified name space and the *command* parameter is set to **not overwrite**, an error is reported to the calling process.

The namespace `sysxrset` is reserved for exclusive *rsets*. When an exclusive *rset* is created, the state of CPUs in the *rset* is modified so that those CPUs only run work that is directed to them. See Exclusive use processor resource sets in *Operating system and device management* and the mkrset command for more information. Root privilege or CAP_NUMA_ATTACH capability is required to create or remove an exclusive *rset*. An exclusive *rset* cannot be overwritten.

Note:

1. Registering a resource set definition can only be done by a process that has root authority or CAP_NUMA_ATTACH capability. CAP_NUMA_ATTACH allows non-root users to create or remove an exclusive *rset*.
2. Overwriting an existing name's registration can be done only by a process that has root authority or write access to this name.

An application registered resource set definition is non-persistent. It does not persist over a system boot.

Both the *namespace* and *rsname* parameters may contain up to 255 characters. They must begin with an ASCII alphanumeric character. Only the period (.), minus (-), and underscore (_) characters can be mixed with ASCII alphanumeric characters within these strings. Moreover, the names are case-sensitive, which means there is a difference between uppercase and lowercase letters in resource set names and name spaces.

Parameters

Item	Description
rset	Specifies a resource set handle of a resource set a name should be registered for.
namespace	Points to a null terminated string corresponding to the name space within which *rsname* will be registered.
rsname	Points to a null terminated string corresponding to the name registered with the setting of the resource set specified by *rset*.

Item	Description
mode	Specifies the bit pattern that determines the created name access permissions. It is constructed by logically ORing one or more of the following values, defined in **rset.h**: • **RS_IRUSR**: Gives read rights to the name's owner • **RS_IWUSR**: Gives write rights to the name's owner • **RS_IRGRP**: Gives read rights to users of the same group as the name's owner • **RS_IWGRP**: Gives write rights to users of the same group as the name's owner • **RS_IROTH**: Gives read rights to others • **RS_IWOTH**: Gives write rights to others Read privilege for a user means that the user can retrieve a resource set definition (by issuing a call to the **rs_getnamedrset** subroutine). Write privilege for a user means that the user can redefine a name (by issuing another call to the **rs_getnamedrset** subroutine).
command	Specifies whether the *rsname* parameter's registration should be overwritten if it already exists in the global repository. This parameter takes one of the following values, defined in **rset.h**: • **RS_REDEFINE**: The *rsname* parameter should be redefined if it already exists in the name space identified by *namespace*. In such a case, the calling process must have write access to *rsname*. • **RS_DEFINE**: The *rsname* parameter should not be redefined if it already exists in the name space identified by *namespace*. If this happens, an error is reported to the calling process

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_registername** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	If one of the following is true: • *rsname* is a null pointer. • *namespace* is a null pointer. • *rsname* or *namespace* points to an invalid name. The name length is 0 or greater than the RSET_NAME_SIZE constant (defined in **rset.h**), or the name contains invalid characters. • *mode* identifies an invalid access rights value. • *command* identifies an invalid command value.
EEXIST	The *command* parameter is set to RS_DEFINE and *rsname* already exists in the global repository within the name space identified by *namespace*.
ENOMEM	There is not enough space to create the data structures related to the registry of this resource set.
EPERM	If one of the following is true: • The *command* parameter is set to RS_REDEFINE and the calling process has neither write access to *rsname* nor root authority . • The calling process has neither the attachment privilege nor root authority. • The *namespace* parameter starts with `sys`. This name space is reserved for system use.

Item	Description
EFAULT	Invalid address, and/or exceptions outside **errno** range.

rs_setnameattr Subroutine

Purpose

Sets the access control information of a resource set definition in the system resource set registry.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_setnameattr (namespace, rsname, command, attr)
char *namespace, *rsname;
unsigned int command;
rs_attributes_t * attr;
```

Description

The **rs_setnameattr** subroutine sets (depending on the *command* value) one or more of the owner, group, or access control information of the system registry resource set definition specified by the *namespace* and *rsname* parameters.

The owner ID and/or group ID and/or access control information of the *rsname* parameter must be supplied in the structure pointed to by the *attr* parameter.

Note:

1. In order to be able to set the attributes of a name, the calling process must have root authority or the attachment privilege and an effective user ID equal to that of the *rsname* parameter's owner.
2. Root authority is required to change the resource set definition owner ID, or to set its group ID outside of the caller's list of groups.

Parameters

Item	Description
namespace	Points to a null terminated string corresponding to the name space within which *rsname* should be found.
rsname	Points to a null terminated string corresponding to the name the information should be retrieved for.
command	Specifies which attributes should be changed. This parameter is constructed by logically ORing one or more of the following values, defined in **rset.h**: • **RS_OWNER**: Set owner as specified in the *owner* field of *attr*. • **RS_GROUP**: Set group as specified in the *group* field of *attr*. • **RS_PERM**: Set access control information as specified in the *mode* field of *attr*.

Item	Description
attr	Points to an **rs_attributes_t** structure containing the *owner, group* and *mode* fields, which will possibly be used by the subroutine for setting attributes. The *mode* field is used to store the access permissions, and is constructed by logically ORing one or more of the following values, defined in **rset.h**: • **RS_IRUSR**: Gives read rights to the name's owner • **RS_IWUSR**: Gives write rights to the name's owner • **RS_IRGRP**: Gives read rights to users of the same group as the name's owner • **RS_IWGRP**: Gives write rights to users of the same group as the name's owner • **RS_IROTH**: Gives read rights to the others • **RS_IWOTH**: Gives write rights to the others

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_setnameattr** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	One of the following is true: • *rsname* is a null pointer. • *namespace* is a null pointer. • *rsname* or *namespace* point to an invalid name. Name length is 0 or greater than the RSET_NAME_SIZE constant (defined in **rset.h**), or name contains invalid characters. • *command* identifies an invalid command value. • *command* includes RS_PERM and the *mode* field of *attr* identifies an invalid access rights value. • *attr* is a null pointer.
EPERM	One of the following is true: • The calling process has neither CAP_NUMA_ATTACH attachment privilege nor root authority. • *command* includes RS_OWNER and the *owner* field of *attr* is different from the caller's user ID and the caller does not have root authority. • *command* includes RS_GROUP, the *group* field of *attr* is outside of the caller's list of groups, and caller does not have root authority. • The *namespace* parameter starts with `sys`. This name space is reserved for system use.
ENOENT	*rsname* could not be found in the name space identified by *namespace*.
ENOSPC	Out of file-space blocks.
EFAULT	Invalid address; exceptions outside **errno** range.
ENOSYS	The **rs_setnameattr** subroutine is not supported by the system.

rs_setpartition Subroutine

Purpose

Sets the partition resource set of a process.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/rset.h>
int rs_setpartition(pid, rset, flags)
pid_t pid;
rsethandle_t rset;
unsigned int flags;
```

Description

The **rs_setpartition** subroutine sets a process' partition resource set. The subroutine can also be used to remove a process' partition resource set.

The partition resource set limits the threads in a process to running only on the processors contained in the partition resource set.

The work component is an existing process identified by the process ID. A process ID value of RS_MYSELF indicates the attachment applies to the current process.

The following conditions must be met to set a process' partition resource set:

- The calling process must have root authority.
- The resource set must contain processors that are available in the system.
- The new partition resource set must be equal to, or a superset of the target process' effective resource set.
- The target process must not contain any threads that have bindprocessor bindings to a processor.
- The resource set must be a superset of all the threads' *rset* in the target process.

The *flags* parameter can be set to indicate the policy for using the resources contained in the resource set specified in the *rset* parameter. The only supported scheduling policy is R_ATTACH_STRSET, which is useful only when the processors of the system are running in simultaneous multithreading mode. Processors like the POWER5 support simultaneous multithreading, where each physical processor has two execution engines, called *hardware threads*. Each hardware thread is essentially equivalent to a single processor, and each is identified as a separate processor in a resource set. The R_ATTACH_STRSET flag indicates that the process is to be scheduled with a single-threaded policy; namely, that it should be scheduled on only one hardware thread per physical processor. If the R_ATTACH_STRSET flag is specified, then all of the available processors indicated in the resource set must be of exclusive use (the processor must belong to some exclusive use processor resource set). A new resource set, called an *ST resource set*, is constructed from the specified resource set and attached to the process according to the following rules:

- All offline processors are ignored.
- If all the hardware threads (processors) of a physical processor (when running in simultaneous multithreading mode, there will be more than one active hardware thread per physical processor) are not included in the specified resource set, the other processors of the processor are ignored when constructing the ST resource set.
- Only one processor (hardware thread) resource per physical processor is included in the ST resource set.

Parameters

Item	Description
pid	Specifies the process ID of the process whose partition resource set is to be set. A value of RS_MYSELF indicates the current process' partition resource set should be set.
rset	Specifies the partition resource set to be set. A value of RS_DEFAULT indicates the process' partition resource set should be removed.
flags	Specifies the policy to use for the process. A value of R_ATTACH_STRSET indicates that the process is to be scheduled with a single-threaded policy (only on one hardware thread per physical processor).

Return Values

If successful, a value of 0 is returned. If unsuccessful, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **rs_setpartition** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EINVAL	The R_ATTACH_STRSET *flags* parameter is specified and one or more processors in the *rset* parameter are not assigned for exclusive use.
ENODEV	The resource set specified by the *rset* parameter does not contain any available processors, or the R_ATTACH_STRSET *flags* parameter is specified and the constructed ST resource set does not have any available processors.
ESRCH	The process identified by the *pid* parameter does not exist.
EFAULT	Invalid address.
ENOMEM	Memory not available.
EPERM	One of the following is true: • The calling process does not have root authority. • The process identified by the *pid* parameter has one or more threads with a bindprocessor processor binding. • The process identified by the *pid* parameter has an effective resource set and the new partition resource set identified by the *rset* parameter does not contain all of the effective resource set's resources. • One of the threads in the process identified by the *pid* parameter has a thread level resource set, and the new partition resource set identified by the *rset* parameter does not contain all of the thread level resource set's resources.

rsqrt Subroutine

Purpose

Computes the reciprocal of the square root of a number.

Libraries

IEEE Math Library (**libm.a**)

System V Math Library (**libmsaa.a**)

Syntax

```
#include <math.h>

double rsqrt(double  x)
```

Description

The **rsqrt** command computes the reciprocal of the square root of a number *x*; that is, 1.0 divided by the square root of *x* (1.0/sqrt(*x*)). On some platforms, using the **rsqrt** subroutine is faster than computing 1.0 / sqrt(*x*). The **rsqrt** subroutine uses the same rounding mode used by the calling program.

When using the **libm.a** library, the **rsqrt** subroutine responds to special values of *x* in the following ways:

- If *x* is NaN, then the **rsqrt** subroutine returns NaN. If *x* is a signaling Nan (NaNS), then the **rsqrt** subroutine returns a quiet NaN and sets the **VX** and **VXSNAN** (signaling NaN invalid operation exception) flags in the FPSCR (Floating-Point Status and Control register) to 1.
- If *x* is +/- 0.0, then the **rsqrt** subroutine returns +/- INF and sets the **ZX** (zero divide exception) flag in the FPSCR to 1.
- If *x* is negative, then the **rsqrt** subroutine returns NaN, sets the **errno** global variable to **EDOM**, and sets the **VX** and **VXSQRT** (square root of negative number invalid operation exception) flags in the FPSCR to 1.

When using the **libmsaa.a** library, the **rsqrt** subroutine responds to special values of *x* in the following ways:

- If *x* is +/- 0.0, then the **rsqrt** subroutine returns +/-HUGE_VAL and sets the **errno** global variable to **EDOM**. The subroutine invokes the **matherr** subroutine, which prints a message indicating a singularity error to standard error output.
- If *x* is negative, then the **rsqrt** subroutine returns 0.0 and sets the **errno** global variable to **EDOM**. The subroutine invokes the **matherr** subroutine, which prints a message indicating a domain error to standard error output.

When compiled with **libmsaa.a**, a program can use the **matherr** subroutine to change these error-handling procedures.

Parameter

Item	Description
x	Specifies a double-precision floating-point value.

Return Values

Upon successful completion, the **rsqrt** subroutine returns the reciprocal of the square root of *x*.

Item	Description
1.0	If *x* is 1.0.
+0.0	If *x* is +INF.

Error Codes

When using either the **libm.a** or **libmsaa.a** library, the **rsqrt** subroutine may return the following error code:

Item	Description
EDOM	The value of *x* is negative.

rstat Subroutines

Purpose

Gets performance data from remote kernels.

Library

(**librpcsvc.a**)

Syntax

```
#include <rpcsvc/rstat.h>
```

```
rstat (host, statp)
char *host;
struct statstime *statp;
```

Description

The **rstat** subroutine gathers statistics from remote kernels. These statistics are available on items such as paging, swapping and CPU utilization.

Parameters

Item	Description
host	Specifies the name of the machine going to be contacted to obtain statistics found in the *statp* parameter.
statp	Contains statistics from *host*.

Return Values

If successful, the **rstat** subroutine fills in the **statstime** for *host* and returns a value of **0**.

Files

Item	Description
/usr/include/rpcsvc/rstat.x	

S

The following Base Operating System (BOS) runtime services begin with the letter *s*.

_showstring Subroutine

Purpose

Dumps the string in the specified string address to the terminal at the specified location.

Library

Curses Library (**libcurses.a**)

Syntax

#include <curses.h>

```
_showstring(Line, Column, First, Last, String)
int  Line,  Column,  First,  Last;
char * String;
```

Description

The **_showstring** subroutine dumps the string in the specified string address to the terminal at the specified location. This is an internal extended curses subroutine and should not normally be called directly by the program.

Parameters

Item	Description
Column	Specifies the horizontal coordinate of the terminal at which to dump the string.
First	Specifies the beginning string address of the string to dump to the terminal.
Last	Specifies the end string address of the string to dump to the terminal.
Line	Specifies the vertical coordinate of the terminal at which to dump the string.
String	Specifies the string to dump to the terminal.

samequantumd32, samequantumd64, or samequantumd128 Subroutine

Purpose

Determines if the representation exponents of both the parameters are the same.

Syntax

```
#include <math.h>

_Bool samequantumd32 (x, y)
_Decimal32 x;
```

```
_Decimal32 y;

_Bool samequantumd64 (x, y)
_Decimal64 x;
_Decimal64 y;

_Bool samequantumd128 (x, y)
_Decimal128 x;
_Decimal128 y;
```

Description

The **samequantumd32**, **samequantumd64**, and **samequantumd128** subroutines determine if the representation exponents of the *x* and *y* parameters are the same. If the values of both the *x* and *y* parameters are NaN, or infinities, they have the same representation exponents; if exactly one operand is infinite, or exactly one operand is NaN, they do not have the same representation exponents. These subroutines raise no exceptions.

Parameters

Item	Description
x	Specifies the value to be computed.
y	Specifies the value to be computed.

Return Values

The **samequantumd32**, **samequantumd64**, and **samequantumd128** subroutines return true when *x* and *y* parameters have the same representation exponents; otherwise false is returned.

savetty Subroutine

Purpose

Saves the state of the tty modes.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
savetty( )
```

Description

The **savetty** subroutine saves the current state of the tty modes in a buffer. It saves the current state in a buffer that the **resetty** subroutine then reads to reset the tty state.

The **savetty** subroutine is called by the **initscr** subroutine and normally should not be called directly by the program.

scalbln, scalblnf, scalblnl, scalbn, scalbnf, scalbnl, or scalb Subroutine

Purpose

Computes the exponent using FLT_RADIX=2.

Syntax

```
#include <math.h>

double scalbln (x, n)
double x;
long n;

float scalblnf (x, n)
float x;
long n;

long double scalblnl (x, n)
long double x;
long n;

double scalbn (x, n)
double x;
int n;

float scalbnf (x, n)
float x;
int n;

long double scalbnl (x, n)
long double x;
int n;

double scalb(x, y)
double x, y;
```

Description

The **scalbln**, **scalblnf**, **scalblnl**, **scalbn**, **scalbnf**, and **scalbnl** subroutines compute *x* * FLT_RADIX^n efficiently, not normally by computing FLT_RADIX^n explicitly. For AIX, FLT_RADIX n=2.

The **scalb** subroutine returns the value of the *x* parameter times 2 to the power of the y parameter.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.
n	Specifies the value to be computed.

Return Values

Upon successful completion, the **scalbln**, **scalblnf**, **scalblnl**, **scalbn**, **scalbnf**, and **scalbnl** subroutines return *x* * FLT_RADIX^n .

If the result would cause overflow, a range error occurs and the **scalbln**, **scalblnf**, **scalblnl**, **scalbn**, **scalbnf**, and **scalbnl** subroutines return ±**HUGE_VAL**, ±**HUGE_VALF**, and ±**HUGE_VALL** (according to the sign of *x*) as appropriate for the return type of the function.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

If *n* is 0, *x* is returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

Error Codes

If the correct value would overflow, the **scalb** subroutine returns +/-INF (depending on a negative or positive value of the x parameter) and sets **errno** to **ERANGE**.

If the correct value would underflow, the **scalb** subroutine returns a value of 0 and sets **errno** to **ERANGE**.

scalblnd32, scalblnd64, scalblnd128, scalbnd32, scalbnd64, or scalbnd128 Subroutine

Purpose

Computes the exponent using FLT_RADIX=10.

Syntax

```
#include <math.h>

_Decimal32 scalblnd32 (x, n)
_Decimal32 x;
long n;

_Decimal64 scalblnd64 (x, n)
_Decimal64 x;
long n;

_Decimal128 scalblnd128 (x, n)
_Decimal128 x;
long n;

_Decimal32 scalbnd32 (x, n)
_Decimal32 x;
int n;

_Decimal64 scalbnd64 (x, n)
_Decimal64 x;
int n;

_Decimal128 scalbnd128 (x, n)
_Decimal128 x;
int n;
```

Description

The **scalblnd32**, **scalblnd64**, **scalblnd128**, **scalbnd32**, **scalbnd64**, and **scalbnd128** subroutines compute *x* * FLT_RADIX^n efficiently, not normally, by computing FLT_RADIX^n explicitly. For AIX, FLT_RADIX =10.

An application checking for error situations must set the value of the **errno** global variable to zero and call the **feclearexcept**(**FE_ALL_EXCEPT**) subroutine before calling any of these subroutines. Upon return, if the value of the **errno** global variable is nonzero or the **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) subroutine is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.
n	Specifies the exponent of 10.

Return Values

Upon successful completion, the **scalblnd32**, **scalblnd64**, **scalblnd128**, **scalbnd32**, **scalbnd64**, and **scalbnd128** subroutines return *x* * FLT_RADIX^{n}.

If the result causes overflow, a range error occurs and the **scalblnd32**, **scalblnd64**, **scalblnd128**, **scalbnd32**, **scalbnd64**, and **scalbnd128** subroutines return ±**HUGE_VAL_D32**, ±**HUGE_VAL_D64**, and ±**HUGE_VAL_D128** (according to the sign of x) as appropriate for the return type of the function.

If the correct value causes underflow and is not representable, a range error occurs and 0.0 is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

If n is 0, x is returned.

If the correct value causes underflow and is representable, a range error occurs and the correct value is returned.

scandir, scandir64, alphasort or alphasort64 Subroutine

Purpose

Scans or sorts directory contents.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/dir.h>

int scandir(DirectoryName,NameList,Select,Compare)
char * DirectoryName;
struct dirent * (* NameList [ ]);
int (* Select) (struct dirent *);
int (* Compare)(void *, void *);

int alphasort ( Directory1,Directory2)
void *Directory1, *Directory2;

int scandir64(DirectoryName,NameList,Select,Compare)
char * DirectoryName;
struct dirent64 * (* NameList [ ]);
int (* Select) (struct dirent64 *);
int (* Compare)(void *, void *);

int alphasort64 ( Directory1,Directory2)
void *Directory1, *Directory2;
```

Description

The **scandir** subroutine reads the directory pointed to by the *DirectoryName* parameter, and then uses the **malloc** subroutine to create an array of pointers to directory entries. The **scandir** subroutine returns the number of entries in the array and, through the *NameList* parameter, a pointer to the array.

The *Select* parameter points to a user-supplied subroutine that is called by the **scandir** subroutine to select which entries to include in the array. The selection routine is passed a pointer to a directory entry and should return a nonzero value for a directory entry that is included in the array. If the *Select* parameter is a null value, all directory entries are included.

The *Compare* parameter points to a user-supplied subroutine. This routine is passed to the **qsort** subroutine to sort the completed array. If the *Compare* parameter is a null value, the array is not sorted. The **alphasort** subroutine provides comparison functions for sorting alphabetically.

The memory allocated to the array can be deallocated by freeing each pointer in the array, and the array itself, with the **free** subroutine.

The **alphasort** subroutine treats *Directory1* and *Directory2* as pointers to **dirent** pointers and alphabetically compares them. This subroutine can be passed as the *Compare* parameter to either the **scandir** subroutine or the **qsort** subroutine, or a user-supplied subroutine can be used.

The `scandir64` subroutine is similar to the `scandir` subroutine except that it returns a pointer to a list of pointers to `struct dirent64` rather than of `struct dirent`.

The `alphasort64` subroutine treats *Directory1* and *Directory2* as pointers to `dirent64` pointers and alphabetically compares them. This subroutine can be passed as the *Compare* parameter to the `scandir64` subroutine, or a user-supplied subroutine can be used.

Parameters

Item	Description
DirectoryName	Points to the directory name.
NameList	Points to the array of pointers to directory entries.
Select	Points to a user-supplied subroutine that is called by the **scandir** subroutine to select which entries to include in the array.
Compare	Points to a user-supplied subroutine that sorts the completed array.
Directory1, Directory2	Point to **dirent** structures for `alphasort`, or to `dirent64` structures for `alphasort64`.

Return Values

The **scandir** subroutine returns the value -1 if the directory cannot be opened for reading or if the **malloc** subroutine cannot allocate enough memory to hold all the data structures. If successful, the **scandir** subroutine returns the number of entries found. If there is no entry inside the directory, the **scandir** subroutine returns 0 and the *Namelist* parameter points to NULL.

The **alphasort** subroutine returns the following values:

Item	Description
Less than 0	The **dirent** structure pointed to by the *Directory1* parameter is lexically less than the **dirent** structure pointed to by the *Directory2* parameter.
0	The **dirent** structures pointed to by the *Directory1* parameter and the *Directory2* parameter are equal.

Item	Description
Greater than 0	The **dirent** structure pointed to by the *Directory1* parameter is lexically greater than the **dirent** structure pointed to by the *Directory2* parameter.

The `scandir64` and `alphasort64` subroutines return the similar values as `scandir` and `alphasort` subroutines, except that returned pointers associated with a `dirent` structure are now associated with a `dirent64` structure.

scanf, fscanf, sscanf, or wsscanf Subroutine

Purpose

Converts formatted input.

Library

Standard C Library (**libc.a**)

or (**libc128.a**)

Syntax

```
#include <stdio.h>

int scanf ( Format [,  Pointer, ... ])
const char *Format;

int fscanf (Stream, Format [, Pointer, ... ])
FILE * Stream;
const char *Format;

int sscanf (String, Format [, Pointer, ... ])
const char * String, *Format;

int wsscanf (wcs, Format [, Pointer, ... ])
const wchar_t *  wcs
const char *Format;
```

Description

The **scanf**, **fscanf**, **sscanf**, and **wsscanf** subroutines read character data, interpret it according to a format, and store the converted results into specified memory locations. If the subroutine receives insufficient arguments for the format, the results are unreliable. If the format is exhausted while arguments remain, the subroutine evaluates the excess arguments but otherwise ignores them.

These subroutines read their input from the following sources:

Item	Description
scanf	Reads from standard input (**stdin**).
fscanf	Reads from the *Stream* parameter.
sscanf	Reads from the character string specified by the *String* parameter.

Item	Description
wsscanf	Reads from the wide character string specified by the *wcs* parameter.

The **scanf**, **fscanf**, **sscanf**, and **wsscanf** subroutines can detect a language-dependent radix character, defined in the program's locale (**LC_NUMERIC**), in the input string. In the C locale, or in a locale that does not define the radix character, the default radix character is a full stop . (period).

Parameters

Item	Description
wcs	Specifies the wide-character string to be read.
Stream	Specifies the input stream.
String	Specifies input to be read.
Pointer	Specifies where to store the interpreted data.
Format	Contains conversion specifications used to interpret the input. If there are insufficient arguments for the *Format* parameter, the results are unreliable. If the *Format* parameter is exhausted while arguments remain, the excess arguments are evaluated as always but are otherwise ignored. The *Format* parameter can contain the following: • Space characters (blank, tab, new-line, vertical-tab, or form-feed characters) that, except in the following two cases, read the input up to the next nonwhite space character. Unless a match in the control string exists, trailing white space (including a new-line character) is not read. • Any character except a % (percent sign), which must match the next character of the input stream. • A conversion specification that directs the conversion of the next input field. The conversion specification consists of the following: – The % (percent sign) or the character sequence %*n*$. **Note:** The %*n*$ character sequence is an X/Open numbered argument specifier. Guidelines for use of the %*n*% specifier are: - The value of *n* in %*n*$ must be a decimal number without leading 0's and must be in the range from 1 to the **NL_ARGMAX** value, inclusive. See the **limits.h** file for more information about the **NL_ARGMAX** value. Using leading 0's (octal numbers) or a larger *n* value can have unpredictable results. - Mixing numbered and unnumbered argument specifications in a format string can have unpredictable results. The only exceptions are %% (two percent signs) and %* (percent sign, asterisk), which can be mixed with the %*n*$ form. - Referencing numbered arguments in the argument list from the format string more than once can have unpredictable results. – The optional assignment-suppression character ***** (asterisk). – An optional decimal integer that specifies the maximum field width. – An optional character that sets the size of the receiving variable for some flags. Use the following optional characters: **l** Long integer rather than an integer when preceding the **d**, **i**, or **n** conversion codes; unsigned long integer rather than unsigned integer when preceding the **o**, **u**, or **x** conversion codes; double rather than float when preceding the **e**, **f**, or **g** conversion codes. **ll** Long long integer rather than an integer when preceding the **d**, **i**, or **n** conversion codes; unsigned long long integer rather than unsigned integer when preceding the **o**, **u**, or **x** conversion codes. **L** A long double rather than a float, when preceding the **e**, **f**, or **g** conversion codes; long integer rather than an integer when preceding the **d**, **i**, or **n** conversion codes; unsigned long integer rather than unsigned integer when preceding the **o**, **u**, or **x** conversion codes. **h** A short integer rather than an integer when preceding the **d**, **i**, and **n** conversion codes; an unsigned short integer (half integer) rather than an unsigned integer when preceding the **o**, **u**, or **x** conversion codes. **H** **_Decimal32** rather than a float, when preceding the **e**, **E**, **f**, **F**, **g**, or **G** conversion codes. **D** **_Decimal64** rather than a float, when preceding the **e**, **E**, **f**, **F**, **g**, or **G** conversion codes. **DD** **_Decimal128** rather than a float, when preceding the **e**, **E**, **f**, **F**, **g**, or **G** conversion codes.

Item	Description
Format (cont.)	• An optional character that sets the size of the receiving variable for vector data types. Use the following optional characters: **v** `vector float` (four 4-byte float components) when preceding the e, E, f, g, G, a, or A conversion codes; `vector signed char` (sixteen 1-byte char components) when preceding the c, d, or i conversion codes; `vector unsigned char` when preceding the o, u, x, or X conversion codes. **vl or lv** `vector signed integer` (four 4-byte integer components) when preceding the d or i conversion codes; `vector unsigned integer` when preceding the o, u, x, or X conversion codes. **vh or hv** `vector signed short` (eight 2-byte integer components) when preceding the d or i conversion codes; `vector unsigned short` when preceding the o, u, x, or X conversion codes. For any of the preceding specifiers, an optional separator character can be specified immediately preceding the vector size specifier. If no separator is specified, the default separator is a space unless the conversion is c, in which case the default separator is null. The set of supported optional separators are , (comma), ; (semicolon), : (colon), and _ (underscore). • A conversion code that specifies the type of conversion to be applied. The conversion specification takes the form: `%[*][width][size]convcode`

The results from the conversion are placed in the memory location designated by the *Pointer* parameter unless you specify assignment suppression with an ***** (asterisk). Assignment suppression provides a way to describe an input field to be skipped. The input field is a string of nonwhite space characters. It extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates how to interpret the input field. The corresponding *Pointer* parameter must be a restricted type. Do not specify the *Pointer* parameter for a suppressed field. You can use the following conversion codes:

%
Accepts a single % (percent sign) input at this point; no assignment or conversion is done. The complete conversion specification should be %% (two percent signs).

d
Accepts an optionally signed decimal integer with the same format as that expected for the subject sequence of the **strtol** subroutine with a value of **10** for the *base* parameter. If no size modifier is specified, the *Pointer* parameter should be a pointer to an integer.

i
Accepts an optionally signed integer with the same format as that expected for the subject sequence of the **strtol** subroutine with a value of **0** for the *base* parameter. If no size modifier is specified, the *Pointer* parameter should be a pointer to an integer.

u
Accepts an optionally signed decimal integer with the same format as that expected for the subject sequence of the **strtoul** subroutine with a value of **10** for the *base* parameter. If no size modifier is specified, the *Pointer* parameter should be a pointer to an unsigned integer.

o

Accepts an optionally signed octal integer with the same format as that expected for the subject sequence of the **strtoul** subroutine with a value of **8** for the *base* parameter. If no size modifier is specified, the *Pointer* parameter should be a pointer to an unsigned integer.

x

Accepts an optionally signed hexadecimal integer with the same format as that expected for the subject sequence of the **strtoul** subroutine with a value of **16** for the *base* parameter. If no size modifier is specified, the *Pointer* parameter should be a pointer to an integer.

e, f, or g

Accepts an optionally signed floating-point number with the same format as that expected for the subject sequence of the **strtod** subroutine. The next field is converted accordingly and stored through the corresponding parameter; if no size modifier is specified, this parameter should be a pointer to a float. The input format for floating-point numbers is a string of digits, with some optional characteristics:

- It can be a signed value.
- It can be an exponential value, containing a decimal rational number followed by an exponent field, which consists of an **E** or an **e** followed by an (optionally signed) integer.
- It can be one of the special values **INF**, **NaNQ**, or **NaNS**. This value is translated into the IEEE-754 value for infinity, quiet **NaN**, or signaling **NaN**, respectively.

p

Matches an unsigned hexadecimal integer, the same as the **%p** conversion of the **printf** subroutine. The corresponding parameter is a pointer to a void pointer. If the input item is a value converted earlier during the same program execution, the resulting pointer compares equal to that value; otherwise, the results of the **%p** conversion are unpredictable.

n

Consumes no input. The corresponding parameter is a pointer to an integer into which the **scanf**, **fscanf**, **sscanf**, or **wsscanf** subroutine writes the number of characters (including wide characters) read from the input stream. The assignment count returned at the completion of this function is not incremented.

s

Accepts a sequence of nonwhite space characters (**scanf**, **fscanf**, and **sscanf** subroutines). The **wsscanf** subroutine accepts a sequence of nonwhite-space wide-character codes; this sequence is converted to a sequence of characters in the same manner as the **wcstombs** subroutine. The *Pointer* parameter should be a pointer to the initial byte of a **char**, signed **char**, or unsigned **char** array large enough to hold the sequence and a terminating null-character code, which is automatically added.

S

Accepts a sequence of nonwhite space characters (**scanf**, **fscanf**, and **sscanf** subroutines). This sequence is converted to a sequence of wide-character codes in the same manner as the **mbstowcs** subroutine. The **wsscanf** subroutine accepts a sequence of nonwhite-space wide character codes. The *Pointer* parameter should be a pointer to the initial wide character code of an array large enough to accept the sequence and a terminating null wide character code, which is automatically added. If the field width is specified, it denotes the maximum number of characters to accept.

c

Accepts a sequence of bytes of the number specified by the field width (**scanf**, **fscanf** and **sscanf** subroutines); if no field width is specified, 1 is the default. The **wsscanf** subroutine accepts a sequence of wide-character codes of the number specified by the field width; if no field width is specified, 1 is the default. The sequence is converted to a sequence of characters in the same manner as the **wcstombs** subroutine. The *Pointer* parameter should be a pointer to the initial bytes of an array large enough to hold the sequence; no null byte is added. The normal skip over white space does not occur.

C

Accepts a sequence of characters of the number specified by the field width (**scanf**, **fscanf**, and **sscanf** subroutines); if no field width is specified, 1 is the default. The sequence is converted to a sequence of wide character codes in the same manner as the **mbstowcs** subroutine. The **wsscanf**

subroutine accepts a sequence of wide-character codes of the number specified by the field width; if no field width is specified, 1 is the default. The *Pointer* parameter should be a pointer to the initial wide character code of an array large enough to hold the sequence; no null wide-character code is added.

[*scanset*]
Accepts a nonempty sequence of bytes from a set of expected bytes specified by the *scanset* variable (**scanf**, **fscanf**, and **sscanf** subroutines). The **wsscanf** subroutine accepts a nonempty sequence of wide-character codes from a set of expected wide-character codes specified by the *scanset* variable. The sequence is converted to a sequence of characters in the same manner as the **wcstombs** subroutine. The *Pointer* parameter should be a pointer to the initial character of a **char**, **signed char**, or **unsigned char** array large enough to hold the sequence and a terminating null byte, which is automatically added. In the **scanf**, **fscanf**, and **sscanf** subroutines, the conversion specification includes all subsequent bytes in the string specified by the *Format* parameter, up to and including the] (right bracket). The bytes between the brackets comprise the *scanset* variable, unless the byte after the [(left bracket) is a ^ (circumflex). In this case, the *scanset* variable contains all bytes that do not appear in the scanlist between the ^ (circumflex) and the] (right bracket). In the **wsscanf** subroutine, the characters between the brackets are first converted to wide character codes in the same manner as the **mbtowc** subroutine. These wide character codes are then used as described above in place of the bytes in the scanlist. If the conversion specification begins with [] or [^], the right bracket is included in the scanlist and the next right bracket is the matching right bracket that ends the conversion specification. You can also:

- Represent a range of characters by the construct *First-Last.* Thus, you can express [0123456789] as [0-9]. The *First* parameter must be lexically less than or equal to the *Last* parameter or else the - (dash) stands for itself. The - also stands for itself whenever it is the first or the last character in the *scanset* variable.
- Include the] (right bracket) as an element of the *scanset* variable if it is the first character of the *scanset*. In this case it is not interpreted as the bracket that closes the *scanset* variable. If the *scanset* variable is an exclusive *scanset* variable, the] is preceded by the ^ (circumflex) to make the] an element of the *scanset*. The corresponding *Pointer* parameter should point to a character array large enough to hold the data field and that ends with a null character (\0). The \0 is added automatically.

A **scanf** conversion ends at the end-of-file (EOF character), the end of the control string, or when an input character conflicts with the control string. If it ends with an input character conflict, the conflicting character is not read from the input stream.

Unless a match in the control string exists, trailing white space (including a new-line character) is not read.

The success of literal matches and suppressed assignments is not directly determinable.

The National Language Support (NLS) extensions to the **scanf** subroutines can handle a format string that enables the system to process elements of the argument list in variable order. The normal conversion character % is replaced by %*n*$, where *n* is a decimal number. Conversions are then applied to the specified argument (that is, the *n*th argument), rather than to the next unused argument.

The first successful run of the **fgetc**, **fgets**, **fread**, **getc**, **getchar**, **gets**, **scanf**, or **fscanf** subroutine using a stream that returns data not supplied by a prior call to the **ungetc** (“ungetc or ungetwc Subroutine” on page 2262) subroutine marks the `st_atime` field for update.

Return Values

These subroutines return the number of successfully matched and assigned input items. This number can be 0 if an early conflict existed between an input character and the control string. If the input ends before the first conflict or conversion, only EOF is returned. If a read error occurs, the error indicator for the stream is set, EOF is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **scanf**, **fscanf**, **sscanf**, and **wsscanf** subroutines are unsuccessful if either the file specified by the *Stream*, *String*, or *wcs* parameter is unbuffered or data needs to be read into the file's buffer and one or more of the following conditions is true:

Item	Description
EAGAIN	The **O_NONBLOCK** flag is set for the file descriptor underlying the file specified by the *Stream*, *String*, or *wcs* parameter, and the process would be delayed in the **scanf**, **fscanf**, **sscanf,** or **wsscanf** operation.
EBADF	The file descriptor underlying the file specified by the *Stream*, *String*, or *wcs* parameter is not a valid file descriptor open for reading.
EINTR	The read operation was terminated due to receipt of a signal, and either no data was transferred or a partial transfer was not reported.

Note: Depending upon which library routine the application binds to, this subroutine may return **EINTR**. Refer to the **signal** ("sigaction, sigvec, or signal Subroutine" on page 1938) subroutine regarding **SA_RESTART**.

Item	Description
EIO	The process is a member of a background process group attempting to perform a read from its controlling terminal, and either the process is ignoring or blocking the **SIGTTIN** signal or the process group has no parent process.
EINVAL	The subroutine received insufficient arguments for the *Format* parameter.
EILSEQ	A character sequence that is not valid was detected, or a wide-character code does not correspond to a valid character.
ENOMEM	Insufficient storage space is available.

scanw, wscanw, mvscanw, or mvwscanw Subroutine

Purpose

Calls the **wgetstr** subroutine on a window and uses the resulting line as input for a scan.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

scanw( Format,  Argument1, Argument2, ...)
char *Format, *Argument1, ...;

wscanw( Window, Format, Argument1, Argument2, ...)
WINDOW *Window;
char *Format, *Argument1, ...;
```

```
mvscanw( Line,  Column, Format, Argument1, Argument2, ...)
int Line, Column;
char *Format, *Argument1, ...;
```

```
mvwscanw(Window, Line, Column, Format, Argument1, Argument2, ...)
WINDOW *Window;
int Line, Column;
char *Format, *Argument1, ...;
```

Description

The **scanw**, **wscanw**, **mvscanw**, and **mvwscanw** subroutines call the **wgetstr** subroutine on a window and use the resulting line as input for a scan. The **mvscanw** and **mvwscanw** subroutines move the cursor before performing the scan function. Use the **scanw** and **mvscanw** subroutines on the stdscr and the **wscanw** and **mvwscanw** subroutines on the user-defined window.

Parameters

Item	Description
Argument	Specifies the input to read.
Column	Specifies the vertical coordinate to move the logical cursor to before performing the scan.
Format	Specifies the conversion specifications to use to interpret the input. For more information about this parameter, see the discussion of the *Format* parameter in the **scanf** (“scanf, fscanf, sscanf, or wsscanf Subroutine” on page 1829) subroutine.
Line	Specifies the horizontal coordinate to move the logical cursor to before performing the scan.
Window	Specifies the window to perform the scan in. You only need to specify this parameter with the **wscanw** and **mvwscanw** subroutines.

Example

The following shows how to read input from the keyboard using the **scanw** subroutine.

```
int id;
char deptname[25];

mvprintw(5,0,"Enter your i.d. followed by the department name:\n");
refresh();
scanw("%d %s", &id, deptname);
mvprintw(7,0,"i.d.: %d, Name: %s\n", id, deptname);
refresh();
```

sched_get_priority_max and sched_get_priority_min Subroutine

Purpose

Retrieves priority limits.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sched.h>

int sched_get_priority_max (policy)
```

```
int policy;

int sched_get_priority_min (policy)
int policy;
```

Description

The **sched_get_priority_max** and **sched_get_priority_min** subroutines return the appropriate maximum or minimum, respectively, for the scheduling policy specified by the *policy* parameter.

The value of the *policy* parameter is one of the scheduling policy values defined in the **sched.h** header file.

Parameters

Item	Description
policy	Specifies the scheduling policy.

Return Values

If successful, the **sched_get_priority_max** and **sched_get_priority_min** subroutines return the appropriate maximum or minimum values, respectively. If unsuccessful, they return -1 and set **errno** to indicate the error.

Error Codes

The **sched_get_priority_max** and **sched_get_priority_min** subroutines fail if:

Item	Description
EINVAL	The value of the *policy* parameter does not represent a defined scheduling policy.
ENOTSUP	This interface does not support processes capable of checkpoint.

sched_getparam Subroutine

Purpose

Gets scheduling parameters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sched.h>

int sched_getparam (pid, param)
pit_t pid;
struct sched_param *param;
```

Description

The **sched_getparam** subroutine returns the scheduling parameters of a process specified by the *pid* parameter in the **sched_param** structure.

If a process specified by the *pid* parameter exists, and if the calling process has permission, the scheduling parameters for the process whose process ID is equal to the value of the *pid* parameter are returned.

If the *pid* parameter is zero, the scheduling parameters for the calling process are returned.

Parameters

Item	Description
pid	Specifies the process for which the scheduling parameters are retrieved.
param	Points to the **sched_param** structure.

Return Values

Upon successful completion, the **sched_getparam** subroutine returns zero. If the **sched_getparam** subroutine is unsuccessful, -1 is returned and **errno** is set to indicate the error.

Error Codes

The **sched_rr_get_interval** subroutine fails if:

Item	Description
EINVAL	The *param* parameter is null or a bad address.
ENOTSUP	This interface does not support processes capable of checkpoint.
EPERM	The requesting process does not have permission to obtain the scheduling parameters of the specified process.
ESRCH	The *pid* parameter is negative, or no process can be found that corresponds to the one specified by the *pid* parameter.

sched_getscheduler Subroutine

Purpose

Gets the scheduling policy.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sched.h>

int sched_getscheduler (pid)
pid_t pid;
```

Description

The **sched_getscheduler** subroutine returns the scheduling policy of the process specified by the *pid* parameter.

The values that can be returned by the **sched_getscheduler** subroutine are defined in the **sched.h** header file.

Parameters

Item	Description
pid	Specifies the process for which the scheduling policy is retrieved.

Return Values

Upon successful completion, the **sched_getscheduler** subroutine returns the scheduling policy of the specified process. If unsuccessful it returns -1 and sets **errno** to indicate the error.

Error Codes

The **sched_getscheduler** subroutine fails if:

Item	Description
EPERM	The requesting process does not have permission to determine the scheduling policy of the specified process.
ESRCH	The *pid* parameter is negative, or no process can be found that corresponds to the one specified by the *pid* parameter.
ENOTSUP	This interface does not support processes capable of checkpoint.

sched_rr_get_interval Subroutine

Purpose

Gets the execution time limits.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sched.h>

int sched_rr_get_interval (pid, interval)
pid_t pid;
struct timespec *interval;
```

Description

The **sched_rr_get_interval** subroutine updates the **timespec** structure referenced by the *interval* parameter to contain the current execution time limit for the process specified by the *pid* parameter.

The current execution time limit applies to process made of system-scope pthreads only, and it is the value of the timeslice tunable for the process specified.

If value of the *pid* parameter is zero, the current execution time limit for the calling process is returned.

Parameters

Item	Description
pid	Specifies the process for which the current execution time limit is retrieved.
interval	Points to the **timespec** structure to be updated.

Return Values

If successful, the **sched_rr_get_interval** subroutine returns zero. Otherwise, it returns -1 and sets **errno** to indicate the error.

Error Codes

The **sched_rr_get_interval** subroutine fails if:

Item	Description
EINVAL	The *param* parameter is null or a bad address.
ENOTSUP	This interface does not support processes capable of checkpoint.
ESRCH	The *pid* parameter is negative, or no process can be found that corresponds to the one specified by the *pid* parameter.

sched_setparam Subroutine

Purpose

Sets scheduling parameters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sched.h>

int sched_setparam (pid, param)
pid_t pid;
const struct sched_param *param;
```

Description

The **sched_setparam** subroutine sets the scheduling parameters of the process specified by the *pid* parameter to the values specified by the **sched_param** structure pointed to by the *param* parameter. The value of the *sched_priority* member in the **sched_param** structure is any integer within the inclusive priority range for the current scheduling policy. Higher numerical values for the priority represent higher priorities.

If a process specified by the *pid* parameter exists, and if the calling process has permission, the scheduling parameters are set for the process whose process ID is equal to the value of the *pid* parameter.

If the *pid* parameter is zero, the scheduling parameters are set for the calling process.

If the caller is favoring a process, it must have SET_PROC_RAC authority. The caller should have the same effective or real user id or BYPASS_DAC_WRITE authority to modify the priority of the process.

Implementations may require the requesting process to have the appropriate authority to set its own scheduling parameters or those of another process.

The target process, whether it is running or not running, is moved to the end of the thread list for its priority.

If the priority of the process specified by the *pid* parameter is set higher than that of the lowest priority running process and if the specified process is ready to run, the process specified by the *pid* parameter preempts the lowest priority running process. Similarly, if the process calling the **sched_setparam** subroutine sets its own priority lower than that of one or more other non-empty process lists, the process that is the head of the highest priority list also preempts the calling process. Thus, the originating process might not receive notification of the completion of the requested priority change until the higher priority process has executed.

Other scheduling policies (such as, SCHED_FIFO2, SCHED_FIFO3, SCHED_FIFO4) behave like fixed priority scheduling policies (such as, SCHED_FIFO and SCHED_RR).

The effect of the **sched_setparam** subroutine on individual threads is dependent on the scheduling contention scope of the threads:

- The **sched_setparam** subroutine has no effect on the scheduling of threads with system scheduling contention scope.
- For threads with process scheduling contention scope, the threads' scheduling parameters are not affected. However, the scheduling of these threads with respect to threads in other processes may be dependent on the scheduling parameters of their process, which are governed using the **sched_setparam** subroutine.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, the underlying kernel-scheduled entities for the system contention scope threads are not affected by the **sched_setparam** subroutine.

The underlying kernel-scheduled entities for the process contention scope threads will have their scheduling parameters changed to the value specified in the *param* parameter. Kernel-scheduled entities for use by process contention scope threads created after this call completes inherit their scheduling policy and associated scheduling parameters from the process.

The **sched_setparam** subroutine is not atomic with respect to other threads in the process. Threads might continue to execute while this subroutine call is in the process of changing the scheduling policy for the underlying kernel-scheduled entities.

Parameters

Item	Description
pid	Specifies the process for which the scheduling parameter is set.
param	Points to the **sched_param** structure.

Return Values

If successful, the **sched_setparam** subroutine returns zero.

If the **sched_setparam** subroutine is unsuccessful, the priority remains unchanged, and the subroutine returns a value of -1 and sets **errno** to indicate the error.

Error Codes

The **sched_setparam** subroutine fails if:

Item	Description
EINVAL	One or more of the requested scheduling parameters is outside the range defined for the scheduling policy of the specified process ID.
EINVAL	The *param* parameter is null or a bad address
ENOTSUP	This interface does not support processes capable of checkpoint.
EPERM	The requesting process does not have permission to set the scheduling parameters for the specified process, or does not have the appropriate authority to invoke the **sched_setparam** subroutine.
ESRCH	The *pid* parameter is negative, or no process can be found that corresponds to the one specified by the *pid* parameter.

sched_setscheduler Subroutine

Purpose

Sets the scheduling policy and parameters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sched.h>

int sched_setscheduler (pid, policy, param)
pid_t pid;
int policy;
const struct sched_param *param;
```

Description

The **sched_setscheduler** subroutine sets the scheduling policy and scheduling parameters of the process specified by the *pid* parameter to the *policy* parameter and the parameters specified in the **sched_param** structure pointed to by *param*, respectively. The value of the *sched_priority* member in the **sched_param** structure is any integer within the inclusive priority range for the scheduling policy.

The possible values for the *policy* parameter are defined in the **sched.h** header file.

If a process specified by the *pid* parameter exists, and if the calling process has permission, the scheduling policy and scheduling parameters are set for the process.

If the *pid* parameter is zero, the scheduling policy and scheduling parameters are set for the calling process.

In order to change a scheduling policy to a fixed priority scheduling policy, the caller must have SET_PROC_RAC authority. When changing the scheduling policy to the SCHED_OTHER scheduling policy, if the former policy was not SCHED_OTHER, the caller must have SET_PROC_RAC authority.

SET_PROC_RAC authority is not needed if the caller wants to defavor a process under the following conditions:

- The *former_policy* process was SCHED_OTHER.
- The new policy is still SCHED_OTHER.
- The new priority is lower than the old priority (the caller wants to defavor the process).
- All the impacted user process-scope threads have a SCHED_OTHER policy.
- The caller should have the same effective or real user id or BYPASS_DAC_WRITE authority.

The **sched_setscheduler** subroutine is successful if it succeeds in setting the scheduling policy and scheduling parameters of the process specified by *pid* to the values specified by the *policy* parameter and the structure pointed to by the *param* parameter, respectively.

The effect of this subroutine on individual threads is dependent on the scheduling contention scope of the following threads:

- The **sched_setscheduler** subroutine has no effect on threads with system scheduling contention scope.
- For threads with process scheduling contention scope, the threads' scheduling policy and associated parameters are not affected. However, the scheduling of these threads with respect to threads in other processes might be dependent on the scheduling parameters of their process, which are governed using the **sched_setscheduler** subroutine.

If an implementation supports a two-level scheduling model in which library threads are multiplexed on top of several kernel-scheduled entities, the underlying kernel-scheduled entities for the system contention scope threads are not affected by these subroutines.

The underlying kernel-scheduled entities for the process contention scope threads have their scheduling policy and associated scheduling parameters changed to the values specified in the *policy* and *param* parameters, respectively. Kernel-scheduled entities for use by process contention scope threads that are created after this call completes inherit their scheduling policy and associated scheduling parameters from the process.

This subroutine is not atomic with respect to other threads in the process. Threads may continue to execute while this subroutine is in the process of changing the scheduling policy and associated scheduling parameters for the underlying kernel-scheduled entities used by the process contention scope threads.

Parameters

Item	Description
pid	Specifies the process for which the scheduling policy and parameters are set.
policy	Contains the scheduling policy and scheduling parameters settings.
param	Points to the **sched_param** structure.

Return Values

Upon successful completion, the **sched_setscheduler** subroutine returns the former scheduling policy of the specified process. If the **sched_setscheduler** subroutine fails to complete successfully, the policy and scheduling parameters will remain unchanged, and the subroutine returns -1 and sets **errno** to indicate the error.

Error Codes

The **sched_setscheduler** subroutine fails if:

Item	Description
EINVAL	The *param* parameter is null or a bad address.
ENOTSUP	This interface does not support processes capable of checkpoint.
EPERM	The requesting process does not have permission to set either or both of the scheduling parameters or the scheduling policy of the specified process.
ESRCH	The *pid* parameter is negative, or no process can be found that corresponds to the one specified by the *pid* parameter.

sched_yield Subroutine

Purpose

Yields the processor.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sched.h>

int sched_yield (void);
```

Description

The **sched_yield** subroutine forces the running thread to relinquish the processor until it again becomes the head of its thread list. It takes no parameters.

Return Values

The **sched_yield** subroutine returns 0 if it completes successfully. Otherwise, it returns -1 and sets **errno** to indicate the error.

Error Codes

The **sched_yield** subroutine fails if:

Item	Description
ENOTSUP	This interface does not support processes capable of checkpoint.

scr_dump, scr_init, scr_restore, scr_set Subroutine

Purpose

File input/output functions.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int scr_dump
(const char *filename);

int scr_init
(const char *filename);

int scr_restore
(const char *filename);

int scr_set
(const char *filename);
```

Description

The **scr_dump** subroutine writes the current contents of the virtual screen to the file named by *filename* in an unspecified format.

The **scr_restore** subroutine sets the virtual screen to the contents of the file named by *filename*, which must have been written using the **scr_dump** subroutine. The next refresh operation restores the screen to the way it looked in the dump file.

The **scr_init** subroutine reads the contents of the file named by *filename* and uses them to initialize the Curses data structures to what the terminal currently has on its screen. The next refresh operation bases any updates of this information, unless either of the following conditions is true:

- The terminal has been written to since the virtual screen was dumped to *filename*.
- The terminfo capabilities rmcup and nrrmc are defined for the current terminal.

The **scr_set** subroutine is a combination of **scr_restore** and **scr_init** subroutines. It tells the program that the information i the file named by filename is what is currently on the screen, and also what the program wants on the screen. This can be thought of as a screen inheritance function.

Parameters

Item	Description
filename	

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

For the **scr_dump** subroutine:

To write the contents of the virtual screen to `/tmp/virtual.dump` file, use:

```
scr_dump("/tmp/virtual.dump");
```

For the **scr_restrore** subroutine:

To restore the contents of the virtual screen from the `/tmp/virtual.dump` file and update the terminal screen, use:

```
scr_restore("/tmp/virtual.dump");
doupdate();
```

scr_init Subroutine

Purpose

Initializes the curses data structures from a dump file.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

scr_init(*Filename***)**
char ***Filename*;**

Description

The **scr_init** subroutine initializes the curses data structures from a dump file. You create dump files with the **scr_dump** subroutine. If the file's data is valid, the next screen update is based on the contents of the file rather than clearing the screen and starting from scratch. The data is invalid if the **terminfo** database boolean capability **nrrmc** is TRUE or the contents of the terminal differ from the contents of the dump file.

Note: If **nrrmc** is TRUE, avoid calling the **putp** subroutine with the **exit_ca_mode** value before calling **scr_init** subroutine in your application.

You can call the **scr_init** subroutine after the **initscr** subroutine to update the screen with the dump file contents. Using the **keypad**, **meta**, **slk_clear**, **curs_set**, **flash**, and **beep** subroutines do not affect the contents of the screen, but cause the terminal's modification time to change.

You can allow more than one process to share screen dumps. Both processes must be run from the same terminal. The **scr_init** subroutine first ensures that the process that created the dump is in sync with the current terminal data. If the modification time of the terminal is not the same as that specified in the dump file, the **scr_init** subroutine assumes that the screen image on the terminal has changed from that in the file, and the file's data is invalid.

If you are allowing two processes to share a screen dump, it is important to understand that one process starts up another process. The following activities happen:

- The second process creates the dump file with the **scr_init** subroutine.
- The second process exits without causing the terminal's time stamp to change by calling the **endwin** subroutine followed by the **scr_dump** subroutine, and then the **exit** subroutine.
- Control is passed back to the first process.
- The first process calls the **scr_init** subroutine to update the screen contents with the dump file data.

Return Values

Item	Description
ERR	Indicates the dump file's time stamp is old or the boolean capability nrrmc is TRUE.
OK	Indicates that the curses data structures were successfully initialized using the contents of the dump file.

Parameters

Item	Description
Filename	Points to a dump file.

scr_restore Subroutine

Purpose

Restores the virtual screen from a dump file.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

scr_restore( FileName)
char *FileName;
```

Description

The **scr_restore** subroutine restores the virtual screen from the contents of a dump file. You create a dump file with the **scr_dump** subroutine. To update the terminal's display with the restored virtual screen, call the **wrefresh** or **doupdate** subroutine after restoring from a dump file.

To communicate the screen image across processes, use the **scr_restore** subroutine along with the **scr_dump** subroutine.

Return Values

Item	Description
ERR	Indicates the content of the dump file is incompatible with the current release of curses.
OK	Indicates that the virtual screen was successfully restored from a dump file.

Parameters

Item	Description
FileName	Identifies the name of the dump file.

Example

To restore the contents of the virtual screen from the `/tmp/virtual.dump` file and update the terminal screen, use:

```
scr_restore("/tmp/virtual.dump");
doupdate();
```

scrl, scroll, wscrl Subroutine

Purpose

Scrolls a Curses window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int scrl
(int n);
```

```
int scroll
(WINDOW *win);
```

```
int wscrl
(WINDOW  *win,
int n);
```

Description

The **scroll** subroutine scrolls win one line in the direction of the first line

The **scrl** and **wscrl** subroutines scroll the current or specified window. If *n* is positive, the window scrolls *n* lines toward the first line. Otherwise, the window scrolls -*n* lines toward the last line.

Theses subroutines do not change the cursor position. If scrolling is disabled for the current or specified window, these subroutines have no effect. The interaction of these subroutines with the **setsccreg** subroutine is currently unspecified.

Parameters

Item	Description
**win*	Specifies the window to scroll.
n	

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

To scroll the user-defined window `my_window` up one line, enter:

```
WINDOW *my_window;
scroll(my_window);
```

scrollok Subroutine

Purpose

Enables or disables scrolling.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

scrollok(*Window*, *Flag***)**
WINDOW ****Window*;**
bool *Flag***;**

Description

The **scrollok** subroutine enables or disables scrolling. Scrolling occurs when a program or user:

- Moves the cursor off the window's bottom edge.
- Enters a new-line character on the last line.
- Types the last character of the last line.

If enabled, **curses** calls a refresh as part of the scrolling action on both the window and the physical display. To get the physical scrolling effect on the terminal, it is also necessary to call the **idlok** (“idlok Subroutine” on page 649) subroutine.

If scrolling is disabled, the cursor is left on the bottom line at the location where the character was entered.

Parameters

Item	Description
Flag	Enables scrolling when set to TRUE. Otherwise, set the *Flag* parameter to FALSE to disable scrolling.
Window	Identifies the window to enable or disable scrolling in.

Examples

1. To turn scrolling on in the user-defined window `my_window`, enter:

```
WINDOW *my_window;
scrollok(my_window, TRUE);
```

2. To turn scrolling off in the user-defined window `my_window`, enter:

```
WINDOW *my_window;
scrollok(my_window, FALSE);
```

sec_getmsgsec Subroutine

Purpose

Gets the security attributes of Interprocess Communication (IPC) message queue.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int sec_getmsgsec (msgid, ipcsec)
int msgid;
ipc_sec_t *ipcsec;
```

Description

The **sec_getmsgsec** subroutine retrieves the security attributes associated with the message queue that is specified by the *msgid* parameter. The returned security attributes are stored in the structure that is pointed to by the *ipcsec* parameter. For a successful completion of the subroutine, the calling process must have MAC and DAC READ access to the message queue.

Parameters

Item	Description
msgid	Specifies the message queue.
ipcsec	Points to an **ipc_sec_t** structure.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EACCES	The calling process does not have permissions or privileges.
EFAULT	The address that the *ipcsec* parameter points to is not valid.
EINVAL	The message queue that the *msgid* parameter specifies is not valid.

sec_getpsec Subroutine

Purpose

Gets the security information that is associated with a process.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/secattr.h>
```

```
int sec_getpsec (pid, credp)
pid_t pid;
secattr_t *credp;
```

Description

The **sec_getpsec**subroutine gets the security attributes structure for the process that is specified by the *pid* parameter. If the value of the *pid* parameter is negative, the information structure of the calling process is retrieved. The *credp* parameter, which is a pointer to an **secattr_t** structure, specifies a buffer holding the security attributes structure to be returned.

Parameters

Item	Description
pid	Specifies the process whose security attributes is to be returned.
credp	Points to the security attribute structure.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EINVAL	The value of the *credp* parameter is NULL or not valid.
ESRCH	No process has a process ID equal to the value of the *pid* parameter.
EPERM	The calling process does not have permissions or privileges.
EFAULT	The address that the *credp* parameter points to is not valid.

sec_getsemsec Subroutine

Purpose

Gets the security attributes of a semaphore identifier.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int sec_getsemsec (semid, ipcsec)
int semid;
ipc_sec_t *ipcsec;
```

Description

The **sec_getsemsec** subroutine retrieves the security attributes associated with the semaphore that is specified by the *semid* parameter. The returned security attributes are stored in the structure that is pointed to by the *ipcsec* parameter. For a successful completion of the subroutine, the calling process must have MAC and DAC READ access to the semaphore.

Parameters

Item	Description
semid	Specifies the semaphore.
ipcsec	Points to an **ipc_sec_t** structure.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EACCES	The calling process does not have permissions or privileges.

Item	Description
EFAULT	The address that the *ipcsec* parameter points to is not valid.
EINVAL	The semaphore that the *semid* parameter specifies is not valid.

sec_getshmsec Subroutine

Purpose

Gets the security attributes of a shared memory segment.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int sec_getshmsec (shmid, ipcsec)
int shmid;
ipc_sec_t *ipcsec;
```

Description

The **sec_getshmsec** subroutine retrieves the security attributes associated with the shared memory segment that is specified by the *shmid* parameter. The returned security attributes are stored in the structure that is pointed to by the *ipcsec* parameter. For a successful completion of the subroutine, the calling process must have MAC and DAC READ access to the shared memory segment.

Parameters

Item	Description
shmid	Specifies the shared memory segment.
ipcsec	Points to an **ipc_sec_t** structure.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EACCES	The calling process does not have permissions or privileges.
EFAULT	The address that the *ipcsec* parameter points to is not valid.
EINVAL	The shared memory segment that the *shmid* parameter specifies is not valid.

sec_getsyslab Subroutine

Purpose

Gets the system sensitivity and integrity labels.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
```

```
int sec_getsyslab (minsl, maxsl, mintl, maxtl)
sl_t *minsl;
sl_t *maxsl;
tl_t *mintl;
tl_t *maxtl;
```

Description

The **sec_getsyslab** subroutine gets the system minimum and maximum sensitivity labels and the system minimum and maximum integrity labels that are being used by the kernel. If the *minsl*, *maxsl*, *mintl*, or *maxtl* parameter is a null pointer, the corresponding label is not retrieved. If the *maxsl* or *maxtl* parameter is requested, either the calling process clearance must dominate the system maximum sensitivity label or integrity label, or the process must have the PV_KER_SECCONFIG or PV_MAC_R privilege.

Parameters

Item	Description
minsl	Points to the minimum sensitivity label.
maxsl	Points to the maximum sensitivity label.
mintl	Points to the minimum integrity label.
maxtl	Points to the maximum integrity label.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EPERM	The calling process does not have permissions or privileges.
EFAULT	The address that the *minsl*, *maxsl*, *mintl*, or *maxtl* parameter points to is not valid.

sec_setmsglab Subroutine

Purpose

Sets the security attributes of an Interprocess Communication (IPC) message queue.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int sec_setmsglab (msgid, sl, tl)
int msgid;
sl_t *sl;
tl_t *tl;
```

Description

The **sec_setmsglab** subroutine sets the security attributes of the message queue that is specified by the *msgid* parameter. The subroutine associates a sensitivity label and an integrity label with the message queue. The *sl* parameter points to the sensitivity label, and the *tl* parameter points to the integrity label. If the *sl* or *tl* parameter is a null pointer, the sensitivity label or integrity label of the message queue remains unchanged.

To change the sensitivity label of a message queue, a process must have the PV_LAB_SL_FILE privilege, DAC and MAC WRITE access to the message queue, and the PV_LAB_SLUG or PV_LAB_SLDG privilege for upgrading or downgrading the label. A process must have DAC OWNER access to the message queue to downgrade the sensitivity label. If the old sensitivity label or the new sensitivity label is outside of the process clearance, the process needs the PV_MAC_CL privilege to change the label.

To change the integrity label of a message queue, a process must have the PV_LAB_TL privilege and have MAC WRITE and DAC OWNER access to the message queue.

Parameters

Item	Description
msgid	Specifies the message queue.
sl	Points to a sensitivity label structure.
tl	Points to an integrity label structure.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EPERM	The calling process does not have permissions or privileges.

Item	Description
EFAULT	The address that the *sl* or *tl* parameter points to is not valid.
EINVAL	The message queue that the *msgid* parameter specifies is not valid.

sec_setplab Subroutine

Purpose

Sets the effective, minimum, and maximum sensitivity labels and the effective, minimum, and maximum integrity labels of a process.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
#include <sys/secconf.h>
```

```
int sec_setplab (pid, eff_sl, mincl, maxcl, eff_tl, min_tl_cl, max_tl_cl)
pid_t pid;
sl_t *eff_sl;
sl_t *mincl;
tl_t *maxcl;
tl_t *eff_tl;
tl_t *min_tl_cl;
tl_t *max_tl_cl;
```

Description

The **sec_setplab** subroutine sets the effective, minimum, and maximum sensitivity labels and the effective, minimum, and maximum integrity labels of the process that is specified by the *pid* parameter.

If the value of the *pid* parameter is negative, the parameters of the calling process are modified.

The calling process and the process being modified must have the same real user ID or the same effective user ID. Or the calling process must have the PV_DAC_O to bypass the user ID restriction.

Effective and Clearance Sensitivity Label

The calling process must have the PV_LAB_SL_SELF privilege to modify its own sensitivity label. The calling process must have the PV_LAB_SL_PROC privilege to modify the sensitivity label of another process.

The effective sensitivity label of the calling process must equal the effective sensitivity label of the target process, or the calling process must have the PV_MAC_W_PROC privilege.

The *eff_sl*, *mincl* and *maxcl* parameters point to the effective, minimum, and maximum sensitivity labels. The maximum sensitivity label must dominate the effective sensitivity label, and the effective sensitivity label must dominate the minimum sensitivity label, if all three labels are specified. If the values of one or more sensitivity label parameters are NULL, the corresponding sensitivity label of the target process is substituted, and the dominance relationship must still be valid. The effective sensitivity label must dominate the current information label of the process being modified. If the effective sensitivity label has a value of NULL, the maximum sensitivity label must dominate the current effective sensitivity label of the process that is specified by the *pid* parameter.

If the effective, minimum, or maximum sensitivity label is outside of the clearance of the calling process, the process must have the PV_MAC_CL privilege.

If the effective, minimum, or maximum sensitivity label results in the corresponding label of the process that is specified by the *pid* parameter being downgraded or upgraded, the process must have the PV_LAB_SL_DG or PV_LAB_SL_UG privilege.

If the *mincl* or *maxcl* parameter is specified, the calling process must have the PV_LAB_CL privilege.

Integrity Label

The PV_LAB_TL privilege is required for a process to set subject or object integrity labels.

The *eff_tl*, *min_tl_cl* and *max_tl_cl* parameters point to the effective, minimum, and maximum integrity labels. The maximum integrity label must dominate the effective integrity label, and the effective integrity label must dominate the minimum integrity label, if all three labels are specified. If the values of one or more integrity label parameters are NULL, the corresponding integrity label of the target process is substituted, and the dominance relationship must still be valid. If the effective integrity label has a value of NULL, the maximum sensitivity label must dominate the current effective integrity label of the process that is specified by the *pid* parameter. If the effective, minimum, or maximum integrity label is outside of the clearance of the calling process, or if the effective integrity label is NOTL; the process must have the PV_MIC_CL privilege.

Neither the *min_tl_cl* nor *max_tl_cl* parameter is allowed to be NOTL. If the *min_tl_cl* or *max_tl_cl* parameter is specified, the calling process must have the PV_LAB_CL_TL privilege.

Parameters

Item	Description
pid	Specifies the process whose security labels are set.
eff_sl	Points to the effective sensitivity label.
mincl	Points to the minimum sensitivity label.
maxcl	Points to the maximum sensitivity label.
eff_tl	Points to the effective integrity label.
min_tl_cl	Points to the minimum integrity label.
max_tl_cl	Points to maximum integrity label.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EINVAL	The values of of all labels arguments that are passed are NULL
ESRCH	No process has a process ID equal to the value of the *pid* parameter.
EPERM	The calling process does not have permissions or privileges.
EFAULT	The address that a label argument points to is not valid.

sec_setsemlab Subroutine

Purpose

Sets the security attributes for a semaphore.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
int sec_setsemlab (semid, sl, tl)
int semid;
sl_t * sl;
tl_t *tl;
```

Description

The **sec_setsemlab** subroutine sets the security attributes of the semaphore that is specified by the *semid* parameter. The subroutine associates a sensitivity label and an integrity label with the semaphore. The *sl* parameter points to the sensitivity label, and the *tl* parameter points to the integrity label. If the *sl* or *tl* parameter is a null pointer, the sensitivity label or integrity label of the semaphore remains unchanged.

To change the sensitivity label of a semaphore, a process must have the PV_LAB_SL_FILE privilege, DAC and MAC WRITE access to the semaphore, and the PV_LAB_SLUG or PV_LAB_SLDG privilege for upgrading or downgrading the label. A process must have DAC OWNER access to the semaphore to downgrade the sensitivity label. If the old sensitivity label or the new sensitivity label is outside of the process clearance, the process needs the PV_MAC_CL privilege to change the label.

To change the integrity label of a semaphore, a process must have the PV_LAB_TL privilege and have MAC WRITE and DAC OWNER access to the semaphore.

Parameters

Item	Description
semid	Specifies the semaphore.
sl	Points to a sensitivity label structure.
tl	Points to an integrity label structure.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EPERM	The calling process does not have permissions or privileges.

Item	Description
EFAULT	The address that the *sl* or *tl* parameter points to is not valid.
EINVAL	The semaphore that the *semid* parameter specifies is not valid.

sec_setshmlab Subroutine

Purpose

Sets the security attributes for a shared memory segment.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

```
int sec_setshmlab (shmid, sl, tl)
int shmid;
sl_t *sl;
tl_t *tl;
```

Description

The **sec_setshmlab** subroutine sets the security attributes of the shared memory segment that is specified by the *shmid* parameter. The subroutine associates a sensitivity label and an integrity label with the shared memory segment. The *sl* parameter points to the sensitivity label, and the *tl* parameter points to the integrity label. If the *sl* or *tl* parameter is a null pointer, the sensitivity label or integrity label of the shared memory segment remains unchanged.

To change the sensitivity label of a shared memory segment, a process must have the PV_LAB_SL_FILE privilege, DAC and MAC WRITE access to the shared memory segment, and the PV_LAB_SLUG or PV_LAB_SLDG privilege for upgrading or downgrading the label. A process must have DAC OWNER access to the shared memory segment to downgrade the sensitivity label. If the old sensitivity label or the new sensitivity label is outside of the process clearance, the process needs the PV_MAC_CL privilege to change the label.

To change the integrity label of a shared memory segment, a process must have the PV_LAB_TL privilege and have MAC WRITE and DAC OWNER access to the shared memory segment.

Parameters

Item	Description
shmid	Specifies the shared memory segment.
sl	Points to a sensitivity label structure.
tl	Points to an integrity label (TL) structure.

Return Values

Item	Description
0	Successful

Item	Description
-1	Unsuccessful

Error Codes

Item	Description
EPERM	The calling process does not have permissions or privileges.
EFAULT	The address that the *sl* or *tl* parameter points to is not valid.
EINVAL	The shared memory segment that the *shmid* parameter specifies is not valid.

sec_setsyslab Subroutine

Purpose

Sets the system sensitivity and integrity labels.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/mac.h>
#include <sys/secconf.h>
```

```
int sec_setsyslab (minsl, maxsl, mintl, maxtl)
sl_t *minsl;
sl_t *maxsl;
tl_t *mintl;
tl_t *maxtl;
```

Description

The **sec_setsyslab** subroutine sets the system minimum and maximum sensitivity labels, and the system minimum and maximum integrity labels to be used by the kernel. If the value a label is not specified, or is NULL, that label will not be changed in the kernel. The calling process must have the PV_KER_SECCONFIG privilege in its effective privilege set.

Parameters

Item	Description
minsl	Points to the minimum sensitivity label.
maxsl	Points to the maximum sensitivity label.
mintl	Points to the minimum integrity label.
maxtl	Points to the maximum integrity label.

Return Values

Item	Description
0	Successful
-1	Unsuccessful

Error Codes

Item	Description
EPERM	The calling process does not have permissions or privileges.
EFAULT	The address that the *minsl*, *maxsl*, *mintl*, or *maxtl* parameter points to is not valid.

select Subroutine

Purpose

Checks the I/O status of multiple file descriptors and message queues.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>
#include <sys/select.h>
#include <sys/types.h>
```

```
int select (Nfdsmsgs, ReadList, WriteList, ExceptList, TimeOut)
int  Nfdsmsgs;
struct sellist * ReadList, *WriteList, *ExceptList;
struct timeval * TimeOut;
```

Description

The **select** subroutine checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or if they have an exceptional condition pending.

When selecting on an unconnected stream socket, select returns when the connection is made. If selecting on a connected stream socket, then the ready message indicates that data can be sent or received. Files descriptors of regular files always select true for read, write, and exception conditions. For more information on sockets, refer to "Understanding Socket Connections" and the related "Checking for Pending Connections Example Program" dealing with pending connections in *AIX Version 6.1 Communications Programming Concepts*.

The **select** subroutine is also supported for compatibility with previous releases of this operating system and with BSD systems.

On shared memory descriptors, the **select** subroutine returns true.

Note: If selecting on a non-blocking socket for both read and write events and if the destination host is unreachable, **select** could show a different behavior due to timing constraints. Refer to the Examples section of this document for further information..

Parameters

Item	Description
Nfdsmsgs	Specifies the number of file descriptors and the number of message queues to check. The low-order 16 bits give the length of a bit mask that specifies which file descriptors to check; the high-order 16 bits give the size of an array that contains message queue identifiers. If either half of the *Nfdsmsgs* parameter is equal to a value of 0, the corresponding bit mask or array is assumed not to be present.

Item	Description

TimeOut	Specifies either a null pointer or a pointer to a **timeval** structure that specifies the maximum length of time to wait for at least one of the selection criteria to be met. The **timeval** structure is defined in the **/usr/include/sys/time.h** file and it contains the following members:
	`struct timeval {`
	`  int tv_sec;  /* seconds  */`
	`  int tv_usec;  /* microseconds  */`
	`  };`
	The number of microseconds specified in *TimeOut*.`tv_usec`, a value from 0 to 999999, is set to one millisecond if the process does not have root user authority and the value is less than one millisecond.
	If the *TimeOut* parameter is a null pointer, the **select** subroutine waits indefinitely, until at least one of the selection criteria is met. If the *TimeOut* parameter points to a **timeval** structure that contains zeros, the file and message queue status is polled, and the **select** subroutine returns immediately.

Item	Description

ReadList, *WriteList*, *ExceptList*	Specify what to check for reading, writing, and exceptions, respectively. Together, they specify the selection criteria. Each of these parameters points to a **sellist** structure, which can specify both file descriptors and message queues. Your program must define the **sellist** structure in the following form:
	`struct sellist`
	`{`
	`ulong fdsmask[F];  /* file descriptor bit mask  */`
	`int msgids[M];  /* message queue identifiers */`
	`};`
	The `fdsmask` array is treated as a bit string in which each bit corresponds to a file descriptor. File descriptor *n* is represented by the bit**(1 << (***n* **mod** *bits***))** in the array element `fdsmask`**[***n* **/ BITS(int)]**. (The **BITS** macro is defined in the **values.h** file.) Each bit that is set to 1 indicates that the status of the corresponding file descriptor is to be checked.
	Note: The low-order 16 bits of the *Nfdsmsgs* parameter specify the number of *bits* (not elements) in the `fdsmask` array that make up the file descriptor mask. If only part of the last int is included in the mask, the appropriate number of low-order bits are used, and the remaining high-order bits are ignored. If you set the low-order 16 bits of the *Nfdsmsgs* parameter to 0, you must *not* define an `fdsmask` array in the **sellist** structure.
	Each int of the `msgids` array specifies a message queue identifier whose status is to be checked. Elements with a value of -1 are ignored. The high-order 16 bits of the *Nfdsmsgs* parameter specify the number of elements in the `msgids` array. If you set the high-order 16 bits of the *Nfdsmsgs* parameter to 0, you must *not* define a `msgids` array in the **sellist** structure.
	Note: The arrays specified by the *ReadList*, *WriteList*, and *ExceptList* parameters are the same size because each of these parameters points to the same **sellist** structure type. However, you need not specify the same number of file descriptors or message queues in each. Set the file descriptor bits that are not of interest to 0, and set the extra elements of the `msgids` array to -1.
	You can use the **SELLIST** macro defined in the **sys/select.h** file to define the **sellist** structure. The format of this macro is:
	SELLIST(*f***,** *m***)** *declarator . . .* **;**
	where *f* specifies the size of the `fdsmask` array, *m* specifies the size of the `msgids` array, and each *declarator* is the name of a variable to be declared as having this type.

Return Values

Upon successful completion, the **select** subroutine returns a value that indicates the total number of file descriptors and message queues that satisfy the selection criteria. The `fdsmask` bit masks are modified so that bits set to 1 indicate file descriptors that meet the criteria. The `msgids` arrays are altered so that message queue identifiers that do not meet the criteria are replaced with a value of -1.

The return value is similar to the *Nfdsmsgs* parameter in that the low-order 16 bits give the number of file descriptors, and the high-order 16 bits give the number of message queue identifiers. These values indicate the sum total that meet each of the read, write, and exception criteria. Therefore, the same file descriptor or message queue can be counted up to three times. You can use the **NFDS** and **NMSGS** macros found in the **sys/select.h** file to separate out these two values from the return value. For example, if *rc* contains the value returned from the **select** subroutine, **NFDS**(*rc*) is the number of files selected, and **NMSGS**(*rc*) is the number of message queues selected.

If the time limit specified by the *TimeOut* parameter expires, the **select** subroutine returns a value of 0.

If a connection-based socket is specified in the *Readlist* parameter and the connection disconnects, the **select** subroutine returns successfully, but the **recv** subroutine on the socket will return a value of 0 to indicate the socket connection has been closed.

For nonbloking connection-based sockets, both successful and unsuccessful connections will cause the **select** subroutine to return successfully without any error.

When the connection completes successfully the socket becomes writable, and if the connection encounters an error the socket becomes both readable and writable.

When using the **select** subroutine, you can not check any pending errors on the socket. You need to call the **getsockopt** subroutine with **SOL_SOCKET** and **SOL_ERROR** to check for a pending error.

If the **select** subroutine is unsuccessful, it returns a value of -1 and sets the global variable **errno** to indicate the error. In this case, the contents of the structures pointed to by the *ReadList*, *WriteList*, and *ExceptList* parameters are unpredictable.

Error Codes

The **select** subroutine is unsuccessful if one of the following are true:

Item	Description
EBADF	An invalid file descriptor or message queue identifier was specified.
EAGAIN	Allocation of internal data structures was unsuccessful.
EINTR	A signal was caught during the **select** subroutine and the signal handler was installed with an indication that subroutines are not to be restarted.
EINVAL	An invalid value was specified for the *TimeOut* parameter or the *Nfdsmsgs* parameter.
EINVAL	The STREAM or multiplexer referenced by one of the file descriptors is linked (directly or indirectly) downstream from a multiplexer.
EFAULT	The *ReadList*, *WriteList*, *ExceptList*, or *TimeOut* parameter points to a location outside of the address space of the process.

Examples

The following is an example of the behavior of the **select** subroutine called on a non-blocking socket, when trying to connect to a host that is unreachable:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <fcntl.h>
#include <sys/time.h>
#include <errno.h>
#include <stdio.h>

int main()
{
    int sockfd, cnt, i = 1;
    struct sockaddr_in  serv_addr;

    bzero((char *)&serv_addr, sizeof (serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr("172.16.55.25");
    serv_addr.sin_port = htons(102);

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        exit(1);
    if (fcntl(sockfd, F_SETFL, FNONBLOCK) < 0)
        exit(1);
    if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof
            (serv_addr)) < 0 && errno != EINPROGRESS)
        exit(1);
```

```
    for (cnt=0; cnt<2; cnt++) {
        fd_set  readfds, writefds;

        FD_ZERO(&readfds);
        FD_SET(sockfd, &readfds);
        FD_ZERO(&writefds);
        FD_SET(sockfd, &writefds);

        if (select(sockfd + 1, &readfds, &writefds, NULL,
                NULL) < 0)
            exit(1);
        printf("Iteration %d ==============\n", i);
        printf("FD_ISSET(sockfd, &readfds) == %d\n",
            FD_ISSET(sockfd, &readfds));
        printf("FD_ISSET(sockfd, &writefds) == %d\n",
            FD_ISSET(sockfd, &writefds));
        i++;
    }
    return 0;
}
```

Here is the output of the above program :

```
Iteration 1 ==============
FD_ISSET(sockfd, &readfds) == 0
FD_ISSET(sockfd, &writefds) == 1
Iteration 2 ==============
FD_ISSET(sockfd, &readfds) == 1
FD_ISSET(sockfd, &writefds) == 1
```

In the first iteration, **select** notifies the write event only. In the second iteration, **select** notifies both the read and write events.

Notes

FD_SETSIZE is the #define variable that defines how many file descriptors the various FD macros will use. The default value for **FD_SETSIZE** is 65534 open file descriptors. This value can not be set greater than **OPEN_MAX**.

For more information, refer to the **/usr/include/sys/time.h** file.

The user may override **FD_SETSIZE** to select a smaller value before including the system header files. This is desirable for performance reasons, because of the overhead in **FD_ZERO** to zero 65534 bits.

Performance Issues and Recommended Coding Practices

The **select** subroutine can be a very compute intensive system call, depending on the number of open file descriptors used and the lengths of the bitmaps used. Do not follow the examples shown in many text books. Most were written when the number of open files supported was small, and thus the bitmaps were short. You should avoid the following (where **select** is being passed **FD_SETSIZE** as the number of FDs to process):

```
select(FD_SETSIZE, ....)
```

Performance will be poor if the program uses **FD_ZERO** and the default **FD_SETSIZE**. **FD_ZERO** should not be used in any loops or before each **select** call. However, using it one time to zero the bit string will not cause problems. If you plan to use this simple programming method, you should override **FD_SETSIZE** to define a smaller number of FDs. For example, if your process will only open two FDs that you will be selecting on, and there will never be more than a few hundred other FDs open in the process, you should lower **FD_SETSIZE** to approximately 1024.

Do not pass **FD_SETSIZE** as the first parameter to **select**. This specifies the maximum number of file descriptors the system should check for. The program should keep track of the highest FD that has been assigned or use the **getdtablesize** subroutine to determine this value. This saves passing excessively long bit maps in and out of the kernel and reduces the number of FDs that **select** must check.

Use the **poll** system call instead of **select**. The **poll** system call has the same functionality as **select**, but it uses a list of FDs instead of a bitmap. Thus, if you are only selecting on a single FD, you would only pass

one FD to **poll**. With **select**, you have to pass a bitmap that is as long as the FD number assigned for that FD. If AIX assigned FD 4000, for example, you would have to pass a bitmap 4001 bits long.

sem_close Subroutine

Purpose

Closes a named semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <semaphore.h>

int sem_close (sem)
sem_t *sem;
```

Description

The **sem_close** subroutine indicates that the calling process is finished using the named semaphore indicated by the *sem* parameter. Calling **sem_close** for an unnamed semaphore (one created by **sem_init**) returns an error. The **sem_close** subroutine deallocates (that is, makes available for reuse by a subsequent calls to the **sem_open** subroutine) any system resources allocated by the system. If the process attempts subsequent uses of the semaphore pointed to by *sem*, an error is returned. If the semaphore has not been removed with a successful call to the **sem_unlink** subroutine, the **sem_close** subroutine has no effect on the state of the semaphore. If the **sem_unlink** subroutine has been successfully invoked for the *name* parameter after the most recent call to **sem_open** with the **O_CREAT** flag set, when all processes that have opened the semaphore close it, the semaphore is no longer accessible.

Parameters

Item	Description
sem	Indicates the semaphore to be closed.

Return Values

Upon successful completion, 0 is returned. Otherwise, -1 is returned and **errno** is set to indicate the error.

Error Codes

The **sem_close** subroutine fails if:

Item	Description
EFAULT	Invalid user address.
EINVAL	The *sem* parameter is not a valid semaphore descriptor.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

sem_destroy Subroutine

Purpose

Destroys an unnamed semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <semaphore.h>

int sem_destroy (sem)
sem_t *sem;
```

Description

The **sem_destroy** subroutine destroys the unnamed semaphore indicated by the *sem* parameter. Only a semaphore that was created using the **sem_init** subroutine can be destroyed using the **sem_destroy** subroutine; calling **sem_destroy** with a named semaphore returns an error. Subsequent use of the semaphore *sem* returns an error until *sem* is reinitialized by another call to **sem_init**. It is safe to destroy an initialized semaphore upon which other threads are currently blocked.

Parameters

Item	Description
sem	Indicates the semaphore to be closed.

Return Values

Upon successful completion, 0 is returned. Otherwise, -1 is returned and **errno** set to indicate the error.

Error Codes

The **sem_destroy** subroutine fails if:

Item	Description
EACCES	Permission is denied to destroy the unnamed semaphore.
EFAULT	Invalid user address.
EINVAL	The *sem* parameter is not a valid semaphore.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

sem_getvalue Subroutine

Purpose

Gets the value of a semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <semaphore.h>

int sem_getvalue (sem, sval)
sem_t *restrict sem;
int *restrict sval;
```

Description

The **sem_getvalue** subroutine updates the location referenced by the *sval* parameter to have the value of the semaphore referenced by the *sem* parameter without affecting the state of the semaphore. The updated value represents an actual semaphore value that occurred at some unspecified time during the call, but it need not be the actual value of the semaphore when it is returned to the calling process.

If the *sem* parameter is locked, the object to which the *sval* parameter points is set to a negative number whose absolute value represents the number of processes waiting for the semaphore at an unspecified time during the call.

Parameters

Item	Description
sem	Indicates the semaphore to be retrieved.
sval	Specifies the location where the semaphore value is stored.

Return Values

Upon successful completion, the **sem_getvalue** subroutine returns a 0. Otherwise, it returns a -1 and sets **errno** to indicate the error.

Error Codes

The **sem_getvalue** subroutine fails if:

Item	Description
EACCES	Permission is denied to access the unnamed semaphore.
EFAULT	Invalid user address.
EINVAL	The *sem* parameter does not refer to a valid semaphore.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

sem_init Subroutine

Purpose

Initializes an unnamed semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <semaphore.h>
```

```
int sem_init (sem, pshared, value)
sem_t *sem;
int pshared;
unsigned value;
```

Description

The **sem_init** subroutine initializes the unnamed semaphore referred to by the *sem* parameter. The value of the initialized semaphore is contained in the *value* parameter. Following a successful call to the **sem_init** subroutine, the semaphore might be used in subsequent calls to the **sem_wait**, **sem_trywait**, **sem_post**, and **sem_destroy** subroutines. This semaphore remains usable until it is destroyed.

If the *pshared* parameter has a nonzero value, the semaphore is shared between processes. In this case, any process that can access the *sem* parameter can use it for performing **sem_wait**, **sem_trywait**, **sem_post**, and **sem_destroy** operations.

Only the *sem* parameter itself may be used for performing synchronization.

If the *pshared* parameter is zero, the semaphore is shared between threads of the process. Any thread in this process can use the *sem* parameter for performing **sem_wait**, **sem_trywait**, **sem_post**, and **sem_destroy** operations. The use of the semaphore by threads other than those created in the same process returns an error.

Attempting to initialize a semaphore that has been already initialized results in the loss of access to the previous semaphore.

Parameters

Item	Description
sem	Specifies the semaphore to be initialized.
pshared	Determines whether the semaphore can be shared between processes or not.
value	Contains the value of the initialized semaphore.

Return Values

Upon successful completion, the **sem_init** subroutine initializes the semaphore in the *sem* parameter. Otherwise, it returns -1 and sets **errno** to indicate the error.

Error Codes

The **sem_init** subroutine fails if:

Item	Description
EFAULT	Invalid user address.
EINVAL	The *value* parameter exceeds SEM_VALUE_MAX.
ENFILE	Too many semaphores are currently open in the system.
ENOMEM	Insufficient memory for the required operation.
ENOSPC	A resource required to initialize the semaphore has been exhausted, or the limit on semaphores, SEM_NSEMS_MAX, has been reached.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

sem_open Subroutine

Purpose

Initializes and opens a named semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <semaphore.h>

sem_t * sem_open (const char *name, int oflag, mode_t mode, unsigned value)
```

Description

The **sem_open** subroutine establishes a connection between a named semaphore and a process. Following a call to the **sem_open** subroutine with semaphore name *name*, the process may reference the semaphore using the address returned from the call. This semaphore may be used in subsequent calls to the **sem_wait**, **sem_trywait**, **sem_post**, and **sem_close** subroutines. The semaphore remains usable by this process until the semaphore is closed by a successful call to **sem_close**, **_exit**, or one of the **exec** subroutines.

The *name* parameter points to a string naming a semaphore object. The name has no representation in the file system. The *name* parameter conforms to the construction rules for a pathname. It might begin with a slash character, and it must contain at least one character. Processes calling **sem_open()** with the same value of *name* refers to the same semaphore object, as long as that name has not been removed.

If a process makes multiple successful calls to the **sem_open** subroutine with the same value of the *name* parameter, the same semaphore address is returned for each such successful call, provided that there have been no calls to the **sem_unlink** subroutine for this semaphore.

Parameters

Item	Description
name	Points to a string naming a semaphore object.

Item	Description
oflag	Controls whether the semaphore is created or merely accessed by the call to the **sem_open** subroutine. The following flag bits may be set in the *oflag* parameter: **O_CREAT** This flag is used to create a semaphore if it does not already exist. If the **O_CREAT** flag is set and the semaphore already exists, the **O_CREAT** flag has no effect, except as noted under the description of the **O_EXCL** flag. Otherwise, the **sem_open** subroutine creates a named semaphore. The **O_CREAT** flag requires a third and a fourth parameter: *mode*, which is of type **mode_t**, and *value*, which is of type **unsigned**. The semaphore is created with an initial value of *value*. Valid initial values for semaphores are less than or equal to **SEM_VALUE_MAX**. The user ID of the semaphore is set to the effective user ID of the process. The group ID of the semaphore is set to the effective group ID of the process. The permission bits of the semaphore are set to the value of the *mode* parameter except those set in the file mode creation mask of the process. When bits in mode other than file permission bits are set, they have no effect. When bits in mode other than file permission bits are set, they have no effect. After the semaphore named *name* has been created by the **sem_open** subroutine with the **O_CREAT** flag, other processes can connect to the semaphore by calling the **sem_open** subroutine with the same value of *name*. **O_EXCL** If the **O_EXCL** and **O_CREAT** flags are set, the **sem_open** subroutine fails if the semaphore name exists. The check for the existence of the semaphore and the creation of the semaphore if it does not exist are atomic with respect to other processes executing the **sem_open** subroutine with the **O_EXCL** and **O_CREAT** flags set. If **O_EXCL** is set and **O_CREAT** is not set, **O_EXCL** is ignored. If flags other than **O_CREAT** and **O_EXCL** are specified in the *oflag* parameter, they have no effect.
mode	Specifies the value of the file permission bits. Used with **O_CREAT** to create a message queue.
value	Specifies the initial value. Used with **O_CREAT** to create a message queue.

Return Values

Upon successful completion, the **sem_open** subroutine returns the address of the semaphore. Otherwise, it returns a value of **SEM_FAILED** and sets **errno** to indicate the error. The **SEM_FAILED** symbol is defined in the **semaphore.h** header file. No successful return from the **sem_open** subroutine returns the value **SEM_FAILED**.

Error Codes

If any of the following conditions occur, the **sem_open** subroutine returns **SEM_FAILED** and sets **errno** to the corresponding value:

Item	Description
EACCES	The named semaphore exists and the permissions specified by *oflag* are denied.
EEXIST	The **O_CREAT** and **O_EXCL** flags are set and the named semaphore already exists.
EFAULT	Invalid user address.
EINVAL	The **sem_open** subroutine is not supported for the given name, or the **O_CREAT** flag was specified in the *oflag* parameter and *value* was greater than **SEM_VALUE_MAX**.

Item	Description
EMFILE	Too many semaphore descriptors are currently in use by this process.
ENAMETOOLONG	The length of the *name* parameter exceeds **PATH_MAX**, or a pathname component is longer than **NAME_MAX**.
ENFILE	Too many semaphores are currently open in the system.
ENOENT	The **O_CREAT** flag is not set and the named semaphore does not exist.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.
ENOSPC	There is insufficient space for the creation of the new named semaphore.

sem_post Subroutine

Purpose

Unlocks a semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <semaphore.h>

int sem_post (sem)
sem_t *sem;
```

Description

The **sem_post** subroutine unlocks the semaphore referenced by the *sem* parameter by performing a semaphore unlock operation on that semaphore.

If the semaphore value resulting from this operation is positive, no threads were blocked waiting for the semaphore to become unlocked, and the semaphore value is incremented.

If the value of the semaphore resulting from this operation is zero, one of the threads blocked waiting for the semaphore is allowed to return successfully from its call to the **sem_wait** subroutine. If the Process Scheduling option is supported, the thread to be unblocked is chosen in a manner appropriate to the scheduling policies and parameters in effect for the blocked threads. In the case of the schedulers SCHED_FIFO and SCHED_RR, the highest priority waiting thread shall be is unblocked, and if there is more than one highest priority thread blocked waiting for the semaphore, then the highest priority thread that has been waiting the longest is unblocked. If the Process Scheduling option is not defined, the choice of a thread to unblock is unspecified.

If the Process Sporadic Server option is supported, and the scheduling policy is SCHED_SPORADIC, the semantics are the same as SCHED_FIFO in the preceding paragraph.

The **sem_post** subroutine is reentrant with respect to signals and may be invoked from a signal-catching function.

Parameters

Item	Description
sem	Specifies the semaphore to be unlocked.

Return Values

If successful, the **sem_post** subroutine returns zero. Otherwise, it returns -1 and sets **errno** to indicate the error.

Error Codes

The **sem_post** subroutine fails if:

Item	Description
EACCES	Permission is denied to access the unnamed semaphore.
EFAULT	Invalid user address.
EIDRM	Semaphore was removed during the required operation.
EINVAL	The *sem* parameter does not refer to a valid semaphore.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

sem_timedwait Subroutine

Purpose

Locks a semaphore (ADVANCED REALTIME).

Syntax

```
#include <semaphore.h>
#include <time.h>

int sem_timedwait(sem_t *restrict sem,
       const struct timespec *restrict abs_timeout);
```

Description

The **sem_timedwait()** function locks the semaphore referenced by *sem* as in the **sem_wait()** function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a **sem_post()** function, this wait terminates when the specified timeout expires.

The timeout expires when the absolute time specified by *abs_timeout* passes—as measured by the clock on which timeouts are based (that is, when the value of that clock equals or exceeds abs_timeout)—or when the absolute time specified by *abs_timeout* has already been passed at the time of the call.

If the **Timers** option is supported, the timeout is based on the CLOCK_REALTIME clock. If the **Timers** option is not supported, the timeout is based on the system clock as returned by the **time()** function. The resolution of the timeout matches the resolution of the clock on which it is based. The **timespec** data type is defined as a structure in the **<time.h>** header.

The function never fails with a timeout if the semaphore can be locked immediately. The validity of the *abs_timeout* parameter does not need to be checked if the semaphore can be locked immediately.

Application Usage

The **sem_timedwait()** function is part of the **Semaphores** and **Timeouts** options and need not be provided on all implementations.

Return Values

The **sem_timedwait()** function returns 0 if the calling process successfully performed the semaphore lock operation on the semaphore designated by *sem*. If the call was unsuccessful, the state of the semaphore remains unchanged, the function returns a value of -1, and *errno* is set to indicate the error.

Error Codes

The **sem_timedwait()** function fails if:

Item	Description
[EFAULT]	*abs_timeout* references invalid memory.
[EINVAL]	The *sem* argument does not refer to a valid semaphore.
[EINVAL]	The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.
[ETIMEDOUT]	The semaphore could not be locked before the specified timeout expired.

The **sem_timedwait()** function might fail if:

Item	Description
[EDEADLK]	A deadlock condition was detected.
[EINTR]	A signal interrupted this function.

sem_trywait and sem_wait Subroutine

Purpose

Locks a semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <semaphore.h>

int sem_trywait (sem)
sem_t *sem;


int sem_wait (sem)
sem_t *sem;
```

Description

The **sem_trywait** subroutine locks the semaphore referenced by the *sem* parameter only if the semaphore is currently not locked; that is, if the semaphore value is currently positive. Otherwise, it does not lock the semaphore.

The **sem_wait** subroutine locks the semaphore referenced by the *sem* parameter by performing a semaphore lock operation on that semaphore. If the semaphore value is currently zero, the calling thread does not return from the call to the **sem_wait** subroutine until it either locks the semaphore or the call is interrupted by a signal.

Upon successful return, the state of the semaphore will be locked and will remain locked until the **sem_post** subroutine is executed and returns successfully.

The **sem_wait** subroutine is interruptible by the delivery of a signal.

Parameters

Item	Description
sem	Specifies the semaphore to be locked.

Return Values

The **sem_trywait** and **sem_wait** subroutines return zero if the calling process successfully performed the semaphore lock operation. If the call was unsuccessful, the state of the semaphore is unchanged, and the subroutine returns -1 and sets **errno** to indicate the error.

Error Codes

The **sem_trywait** and **sem_wait** subroutines fail if:

Item	Description
EACCES	Permission is denied to access the unnamed semaphore.
EAGAIN	The semaphore was already locked, so it cannot be immediately locked by the **sem_trywait** subroutine.
EFAULT	Invalid user address.
EIDRM	Semaphore was removed during the required operation.
EINTR	A signal interrupted the subroutine.
EINVAL	The *sem* parameter does not refer to a valid semaphore.
ENOMEM	Insufficient memory for the required operation.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

sem_unlink Subroutine

Purpose

Removes a named semaphore.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <semaphore.h>

int sem_unlink (name)
const char *name;
```

Description

The **sem_unlink** subroutine removes the semaphore named by the string *name*.

If the semaphore named by *name* is currently referenced by other processes, then **sem_unlink** has no effect on the state of the semaphore. If one or more processes have the semaphore open when **sem_unlink** is called, destruction of the semaphore is postponed until all references to the semaphore have been destroyed by calls to **sem_close**, **_exit**, or **exec**. Calls to **sem_open** to recreate or reconnect to the semaphore refer to a new semaphore after **sem_unlink** is called.

The **sem_unlink** subroutine does not block until all references have been destroyed, and it returns immediately.

Parameters

Item	Description
name	Specifies the name of the semaphore to be unlinked.

Return Values

Upon successful completion, the **sem_unlink** subroutine returns a 0. Otherwise, the semaphore remains unchanged, -1 is returned, and **errno** is set to indicate the error.

Error Codes

The **sem_unlink** subroutine fails if:

Item	Description
EACCES	Permission is denied to unlink the named semaphore.
EFAULT	Invalid user address.
ENAMETOOLONG	The length of the *name* parameter exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
ENOENT	The named semaphore does not exist.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

semctl Subroutine

Purpose

Controls semaphore operations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/sem.h>

int semctl (SemaphoreID, SemaphoreNumber, Command, arg)
OR
int semctl (SemaphoreID, SemaphoreNumber, Command)

int  SemaphoreID;
```

```
int  SemaphoreNumber;
int  Command;
union semun {
         int val;
         struct semid_ds *buf;
         unsigned short *array;
} arg;
```

If the fourth argument is required for the operation requested, it must be of type union semun and explicitly declared as shown above.

Description

The **semctl** subroutine performs a variety of semaphore control operations as specified by the *Command* parameter.

The following limits apply to semaphores:

- Maximum number of semaphore IDs is 131072.
- Maximum number of semaphores per ID is 65,535.
- Maximum number of operations per call by the **semop** ("semop and semtimedop Subroutines" on page 1880) subroutine is 1024.
- Maximum number of undo entries per procedure is 1024.
- Maximum semaphore value is 32,767.
- Maximum adjust-on-exit value is 16,384.

Parameters

SemaphoreID
Specifies the semaphore identifier.

SemaphoreNumber
Specifies the semaphore number.

arg.val
Specifies the value for the semaphore for the **SETVAL** command.

arg.buf
Specifies the buffer for status information for the **IPC_STAT** and **IPC_SET** commands.

arg.array
Specifies the values for all the semaphores in a set for the **GETALL** and **SETALL** commands.

Command
Specifies semaphore control operations.

The following *Command* parameter values are executed with respect to the semaphore specified by the *SemaphoreID* and *SemaphoreNumber* parameters. These operations get and set the values of a **sem** structure, which is defined in the **sys/sem.h** file.

GETVAL
Returns the **semval** value, if the current process has read permission.

SETVAL
Sets the **semval** value to the value specified by the *arg.val* parameter, if the current process has write permission. When this *Command* parameter is successfully executed, the **semadj** value corresponding to the specified semaphore is cleared in all processes.

GETPID
Returns the value of the `sempid` field, if the current process has read permission.

GETNCNT
Returns the value of the `semncnt` field, if the current process has read permission.

GETZCNT
Returns the value of the `semzcnt` field, if the current process has read permission.

The following *Command* parameter values return and set every **semval** value in the set of semaphores. These operations get and set the values of a **sem** structure, which is defined in the **sys/sem.h** file.

GETALL
Stores **semvals** values into the array pointed to by the *arg.array* parameter, if the current process has read permission.

SETALL
Sets **semvals** values according to the array pointed to by the *arg.array* parameter, if the current process has write permission. When this *Command* parameter is successfully executed, the **semadj** value corresponding to each specified semaphore is cleared in all processes.

The following *Commands* parameter values get and set the values of a **semid_ds** structure, defined in the **sys/sem.h** file. These operations get and set the values of a **sem** structure, which is defined in the **sys/sem.h** file.

IPC_STAT
Obtains status information about the semaphore identified by the *SemaphoreID* parameter. This information is stored in the area pointed to by the *arg.buf* parameter.

IPC_SET
Sets the owning user and group IDs, and the access permissions for the set of semaphores associated with the *SemaphoreID* parameter. The **IPC_SET** operation uses as input the values found in the *arg.buf* parameter structure.

IPC_SET sets the following fields:

Item	Description
sem_perm.uid	User ID of the owner
sem_perm.gid	Group ID of the owner
sem_perm.mode	Permission bits only
sem_perm.cuid	Creator's user ID

IPC_SET can only be executed by a process that has root user authority or an effective user ID equal to the value of the `sem_perm.uid` or `sem_perm.cuid` field in the data structure associated with the *SemaphoreID* parameter.

IPC_RMID
Removes the semaphore identifier specified by the *SemaphoreID* parameter from the system and destroys the set of semaphores and data structures associated with it. This *Command* parameter can only be executed by a process that has root user authority or an effective user ID equal to the value of the `sem_perm.uid` or `sem_perm.cuid` field in the data structure associated with the *SemaphoreID* parameter.

Return Values

Upon successful completion, the value returned depends on the *Command* parameter as follows:

Command	Return Value
GETVAL	Returns the value of the `semval` field.
GETPID	Returns the value of the `sempid` field.
GETNCNT	Returns the value of the `semncnt` field.
GETZCNT	Returns the value of the `semzcnt` field.
All Others	Return a value of 0.

If the **semctl** subroutine is unsuccessful, a value of -1 is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **semctl** subroutine is unsuccessful if any of the following is true:

Item	Description
EINVAL	The *SemaphoreID* parameter is not a valid semaphore identifier.
EINVAL	The *SemaphoreNumber* parameter is less than 0 or greater than or equal to the **sem_nsems** value.
EINVAL	The *Command* parameter is not a valid command.
EACCES	The calling process is denied permission for the specified operation.
ERANGE	The *Command* parameter is equal to the **SETVAL** or **SETALL** value and the value to which **semval** value is to be set is greater than the system-imposed maximum.
EPERM	The *Command* parameter is equal to the **IPC_RMID** or **IPC_SET** value and the calling process does not have root user authority or an effective user ID equal to the value of the `sem_perm.uid` or `sem_perm.cuid` field in the data structure associated with the *SemaphoreID* parameter.
EFAULT	The *arg.buf* or *arg.array* parameter points outside of the allocated address space of the process.
ENOMEM	The system does not have enough memory to complete the subroutine.

semget Subroutine

Purpose

Gets a set of semaphores.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/sem.h>

int semget (Key, NumberOfSemaphores, SemaphoreFlag)
key_t  Key;
int  NumberOfSemaphores,  SemaphoreFlag;
```

Description

The **semget** subroutine returns the semaphore identifier associated with the *Key* parameter value.

The **semget** subroutine creates a data structure for the semaphore ID and an array containing the *NumberOfSemaphores* parameter semaphores if one of the following conditions is true:

- The *Key* parameter is equal to the **IPC_PRIVATE** operation.
- The *Key* parameter does not already have a semaphore identifier associated with it, and the **IPC_CREAT** value is set.

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

- The `sem_perm.cuid` and `sem_perm.uid` fields are set equal to the effective user ID of the calling process.
- The `sem_perm.cgid` and `sem_perm.gid` fields are set equal to the effective group ID of the calling process.
- The low-order 9 bits of the `sem_perm.mode` field are set equal to the low-order 9 bits of the *SemaphoreFlag* parameter.
- The `sem_nsems` field is set equal to the value of the *NumberOfSemaphores* parameter.
- The `sem_otime` field is set equal to 0 and the `sem_ctime` field is set equal to the current time.

The data structure associated with each semaphore in the set is not initialized. The **semctl** ("semctl Subroutine" on page 1874) subroutine (with the *Command* parameter values **SETVAL** or **SETALL**) can be used to initialize each semaphore.

If the *Key* parameter value is not **IPC_PRIVATE**, the **IPC_EXCL** value is not set, and a semaphore identifier already exists for the specified *Key* parameter, the value of the *NumberOfSemaphores* parameter specifies the number of semaphores that the current process needs.

If the *NumberOfSemaphores* parameter has a value of 0, any number of semaphores is acceptable. If the *NumberOfSemaphores* parameter is not 0, the **semget** subroutine is unsuccessful if the set contains fewer than the value of the *NumberOfSemaphores* parameter.

The following limits apply to semaphores:

- Maximum number of semaphore IDs 1048576.
- Maximum number of semaphores per ID is 65,535.
- Maximum number of operations per call by the **semop** subroutine is 1024.
- Maximum number of undo entries per procedure is 1024.
- Maximum semaphore value is 32,767.
- Maximum adjust-on-exit value is 16,384.

Parameters

Item	Description
Key	Specifies either the **IPC_PRIVATE** value or an IPC key constructed by the **ftok** subroutine (or a similar algorithm).
NumberOfSemaphores	Specifies the number of semaphores in the set.

Item	Description
SemaphoreFlag	Constructed by logically ORing one or more of the following values: **IPC_CREAT** Creates the data structure if it does not already exist. **IPC_EXCL** Causes the **semget** subroutine to fail if the **IPC_CREAT** value is also set and the data structure already exists. **S_IRUSR** Permits the process that owns the data structure to read it. **S_IWUSR** Permits the process that owns the data structure to modify it. **S_IRGRP** Permits the group associated with the data structure to read it. **S_IWGRP** Permits the group associated with the data structure to modify it. **S_IROTH** Permits others to read the data structure. **S_IWOTH** Permits others to modify the data structure. Values that begin with the **S_I** prefix are defined in the **sys/mode.h** file and are a subset of the access permissions that apply to files.

Return Values

Upon successful completion, the **semget** subroutine returns a semaphore identifier. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **semget** subroutine is unsuccessful if one or more of the following conditions is true:

Item	Description
EACCES	A semaphore identifier exists for the *Key* parameter but operation permission, as specified by the low-order 9 bits of the *SemaphoreFlag* parameter, is not granted.
EINVAL	A semaphore identifier does not exist and the *NumberOfSemaphores* parameter is less than or equal to a value of 0, or greater than the system-imposed value.
EINVAL	A semaphore identifier exists for the *Key* parameter, but the number of semaphores in the set associated with it is less than the value of the *NumberOfSemaphores* parameter and the *NumberOfSemaphores* parameter is not equal to 0.
ENOENT	A semaphore identifier does not exist for the *Key* parameter and the **IPC_CREAT** value is not set.
ENOSPC	Creating a semaphore identifier would exceed the maximum number of identifiers allowed systemwide.
EEXIST	A semaphore identifier exists for the *Key* parameter, but both the **IPC_CREAT** and **IPC_EXCL** values are set.
ENOMEM	There is not enough memory to complete the operation.

semop and semtimedop Subroutines

Purpose

Performs semaphore operations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/sem.h>
```

```
int semop (SemaphoreID, SemaphoreOperations, NumberOfSemaphoreOperations)
int  SemaphoreID;
struct sembuf * SemaphoreOperations;
size_t  NumberOfSemaphoreOperations;
```

```
#include <sys/sem.h>
```

```
int semtimedop (SemaphoreID, SemaphoreOperations,
 NumberOfSemaphoreOperations, Timeout)
int  SemaphoreID;
struct sembuf * SemaphoreOperations;
size_t  NumberOfSemaphoreOperations;
struct timespec * timeout;
```

Description

The **semop** and **semtimedop** subroutines perform operations on the set of semaphores associated with the semaphore identifier specified by the *SemaphoreID* parameter.

The **semtimedop** subroutine limits the time the caller will sleep while waiting for the semaphore operation(s) to complete. The **timespec** structure is defined in the **/usr/include/sys/time.h** file and includes the following fields:

Item	Description
tv_sec	Seconds on timer
tv_nsec	Nanoseconds on timer

If the caller sleeps for the time allotted by the **timespec** structure before the operation(s) can be completed, the current operation is aborted and the **semtimedop** subroutine will return an error.

Note: The **semtimedop** subroutine is available beginning with AIX Version 6.1.

The **sembuf** structure is defined in the **usr/include/sys/sem.h** file. Each **sembuf** structure specified by the *SemaphoreOperations* parameter includes the following fields:

Item	Description
sem_num	Semaphore number
sem_op	Semaphore operation
sem_flg	Operation flags

Each semaphore operation specified by the sem_op field is performed on the semaphore specified by the *SemaphoreID* parameter and the sem_num field. Semaphore operations are performed in the order they are received in the **sembuf** array. The sem_op field specifies one of three semaphore operations.

1. If the sem_op field is a negative integer and the calling process has permission to alter, one of the following conditions occurs:
   - If the **semval** variable (see the **/usr/include/sys/sem.h** file) is greater than or equal to the absolute value of the sem_op field, the absolute value of the sem_op field is subtracted from the **semval** variable. In addition, if the **SEM_UNDO** flag is set in the sem_flg field, the absolute value of the sem_op field is added to the **semadj** value of the calling process for the specified semaphore.
   - If the **semval** variable is less than the absolute value of the sem_op field and the **IPC_NOWAIT** value is set in the sem_flg field, the **semop** or **semtimedop** subroutine returns immediately.
   - If the **semval** variable is less than the absolute value of the sem_op field and the **IPC_NOWAIT** value is not set in the sem_flg field, the **semop** and **semtimedop** subroutine increments the semncnt field associated with the specified semaphore and suspends the calling process until one of the following conditions occurs:
     - The value of the **semval** variable becomes greater than or equal to the absolute value of the sem_op field. The value of the semncnt field associated with the specified semaphore is then decremented, and the absolute value of the sem_op field is subtracted from the **semval** variable. In addition, if the **SEM_UNDO** flag is set in the sem_flg field, the absolute value of the sem_op field is added to the **semadj** value of the calling process for the specified semaphore.
     - The *SemaphoreID* parameter for which the calling process is awaiting action is removed from the system. When this occurs, the **errno** global variable is set to the **EIDRM** flag and a value of -1 is returned.
     - The calling process received a signal that is to be caught. When this occurs, the **semop** and **semtimedop** subroutine decrements the value of the semncnt field associated with the specified semaphore. When the semzcnt field is decremented, the calling process resumes as prescribed by the **sigaction** (“sigaction, sigvec, or signal Subroutine” on page 1938) subroutine.
     - The calling process sleeps for the time allotted by the **timespec** structure. When this occurs, the **errno** global variable is set to the **ETIMEDOUT** flag and a value of -1 is returned.
2. If the sem_op field is a positive integer and the calling process has alter permission, the value of the sem_op field is added to the **semval** variable. In addition, if the **SEM_UNDO** flag is set in the sem_flg field, the value of the sem_op field is subtracted from the calling process's **semadj** value for the specified semaphore.
3. If the value of the sem_op field is 0 and the calling process has read permission, one of the following occurs:
   - If the **semval** variable is 0, the **semop** or **semtimedop** subroutine returns immediately.
   - If the **semval** variable is not equal to 0 and **IPC_NOWAIT** value is set in the sem_flg field, the **semop** or **semtimedop** subroutine returns immediately.
   - If the **semval** variable is not equal to 0 and the **IPC_NOWAIT** value is not set in the sem_flg field, the **semop** or **semtimedop** subroutine increments the semzcnt field associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
     - The value of the **semval** variable becomes 0. When this occurs, the value of the semzcnt field associated with the specified semaphore is decremented.
     - The *SemaphoreID* parameter for which the calling process is awaiting action is removed from the system. If this occurs, the **errno** global variable is set to the **EIDRM** error code and a value of -1 is returned.
     - The calling process received a signal that is to be caught. When this occurs, the **semop** or **semtimedop** subroutine decrements the value of the semzcnt field associated with the specified semaphore. When the semzcnt field is decremented, the calling process resumes execution as prescribed by the **sigaction** subroutine.

– The calling process sleeps for the time allotted by the **timespec** structure. When this occurs, the **errno** global variable is set to the **ETIMEDOUT** flag and a value of -1 is returned.

Note: Calling the **semtimedop** subroutine with an invalid *Timeout* parameter will prevent the calling process from being suspended if necessary. If the *Timeout* parameter specified to the **semtimedop** subroutine is not valid and the calling process needs to be suspended, then the **errno** global variable will be set to indicate the error and a value of -1 will be returned.

The following limits apply to semaphores:

- Maximum number of semaphore IDs is 131072.
- Maximum number of semaphores per ID is 65,535.
- Maximum number of operations per call by the **semop** subroutine is 1024.
- Maximum number of undo entries per procedure is 1024.
- Maximum capacity of a semaphore value is 32,767 bytes.
- Maximum adjust-on-exit value is 16,384 bytes.

Parameters

Item	Description
SemaphoreID	Specifies the semaphore identifier.
NumberOfSemaphoreOperations	Specifies the number of structures in the array.
SemaphoreOperations	Points to an array of structures, each of which specifies a semaphore operation.
Timeout	Points to a structure specifying an interval of time beyond which the operation should not sleep.

Return Values

Upon successful completion, the **semop** and **semtimedop** subroutines return a value of 0. Also, the *SemaphoreID* parameter value for each semaphore that is operated upon is set to the process ID of the calling process.

If the **semop** or **semtimedop** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error. If the **SEM_ORDER** flag was set in the `sem_flg` field for the first semaphore operation in the *SemaphoreOperations* array, the **SEM_ERR** value is set in the `sem_flg` field for the unsuccessful operation.

If the *SemaphoreID* parameter for which the calling process is awaiting action is removed from the system, the **errno** global variable is set to the **EIDRM** error code and a value of -1 is returned.

Error Codes

The **semop** or **semtimedop** subroutine is unsuccessful if one or more of the following are true for any of the semaphore operations specified by the *SemaphoreOperations* parameter. If the operations were performed individually, the discussion of the **SEM_ORDER** flag provides more information about error situations.

Item	Description
EINVAL	The *SemaphoreID* parameter is not a valid semaphore identifier.
EINVAL	The number of individual semaphores for which the calling process requests a **SEM_UNDO** flag would exceed the limit.
EINVAL	The *Timeout* parameter specified a **tv_sec** or **tv_nsec** value less than 0, or a **tv_nsec** value greater than 1000 million.

Item	Description
EFBIG	The `sem_num` value is less than 0 or it is greater than or equal to the number of semaphores in the set associated with the *SemaphoreID* parameter.
E2BIG	The *NumberOfSemaphoreOperations* parameter is greater than the system-imposed maximum.
EACCES	The calling process is denied permission for the specified operation.
EAGAIN	The operation would result in suspension of the calling process, but the **IPC_NOWAIT** value is set in the `sem_flg` field.
ENOMEM	Insufficient memory for the required operation.
ENOSPC	The limit on the number of individual processes requesting a **SEM_UNDO** flag would be exceeded.
EINVAL	The number of individual semaphores for which the calling process requests a **SEM_UNDO** flag would exceed the limit.
ERANGE	An operation would cause a **semval** value to overflow the system-imposed limit.
ERANGE	An operation would cause a **semadj** value to overflow the system-imposed limit.
EFAULT	The *SemaphoreOperations* parameter points outside of the address space of the process.
EINTR	A signal interrupted the **semop** subroutine.
EIDRM	The semaphore identifier *SemaphoreID* parameter has been removed from the system.
EFAULT	The *Timeout* parameter points to an invalid address.
ETIMEDOUT	The time specified by the *Timeout* parameter expired before the requested operations could be completed.

set_curterm Subroutine

Purpose

Sets the current terminal variable to the specified terminal.

Library

Curses Library (**libcurses.a**)

Curses Syntax

```
#include <curses.h>
#include <term.h>
```

set_curterm(*Newterm***)**
TERMINAL * *Newterm***;**

Description

The **cur_term** subroutine sets the **cur_term** variable to the terminal specified by the *Newterm* parameter. The **cur_term** subroutine is useful when the **setupterm** subroutine is called more than once. The **set_curterm** subroutine allows the programmer to toggle back and forth between terminals.

When information for a particular terminal is no longer required, remove it using the **del_curterm** subroutine.

Note: The **cur_term** subroutine is a low-level subroutine. You should use this subroutine only if your application must deal directly with the **terminfo** database to handle certain terminal capabilities. For example, use this subroutine if your application programs function keys.

Parameters

Item	Description
Newterm	Points to a **TERMINAL** structure. This structure contains information about a specific terminal.

Examples

To set the **cur_term** variable to point to the `my_term` terminal, use:

```
TERMINAL *newterm;
set_curterm(newterm);
```

set_term Subroutine

Purpose

Switches between screens.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

SCREEN *set_term
(SCREEN *new);
```

Description

The **set_term** subroutine switches between different screens. The *new* argument specifies the current screen.

Parameters

Item	Description
**new*	

Return Values

Upon successful completion, the **set_term** subroutine returns a pointer to the previous screen. Otherwise, it returns a null pointer.

Examples

To make the terminal stored in the user-defined **SCREEN** variable `my_terminal` the current terminal and then store a pointer to the old terminal in the user-defined variable `old_terminal`, enter:

```
SCREEN *old_terminal, *my_terminal;
old_terminal = set_term(my_terminal);
```

setacldb or endacldb Subroutine

Purpose

Opens and closes the SMIT ACL database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int setacldb(Mode)
int Mode;
```

```
int endacldb;
```

Description

These functions may be used to open and close access to the user SMIT ACL database. Programs that call the **getusraclattr** or **getgrpaclattr** subroutines should call the **setacldb** subroutine to open the database and the **endacldb** subroutine to close the database.

The **setacldb** subroutine opens the database in the specified mode, if it is not already open. The open count is increased by 1.

The **endacldb** subroutine decreases the open count by 1 and closes the database when this count goes to 0. Any uncommitted changed data is lost.

Parameters

Item	Description
Mode	Specifies the mode of the open. This parameter may contain one or more of the following values defined in the **usersec.h** file: **S_READ** Specifies read access. **S_WRITE** Specifies update access.

Return Values

The **setacldb** and **endacldb** subroutines return a value of 0 to indicate success. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setacldb** subroutine fails if the following is true:

Item	Description
EACCES	Access permission is denied for the data request.

Both subroutines return errors from other subroutines.

Security

Security Files Accessed: The calling process must have access to the SMIT ACL data.

Mode File **rw/etc/security/smitacl.user**

setauthdb or setauthdb_r Subroutine

Purpose

Defines the current administrative domain.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <usersec.h>

int setauthdb (New, Old)
authdb_t *New;
authdb_t *Old;

int setauthdb_r (New, Old)
authdb_t *New;
authdb_t *Old;
```

Description

The **setauthdb** and **setauthdb_r** subroutines set the value of the current administrative domain in the ***New*** parameter. The **setauthdb** subroutine sets the value of the current process-wide administrative domain. The **setauthdb_r** subroutine sets the administrative domain for the current thread if one is set. The subroutines return **-1** if no administrative domain is set. The current administrative domain is returned in the ***Old*** parameter. The ***Old*** parameter can be a null pointer if the value of the current administrative domain is not wanted.

The administrative domain determines which user and group information databases are queried by the user and group library functions. The default behavior is to access all of the defined administrative domains. The **setauthdb** subroutine restricts the user and group library functions to the named administrative domains for all threads in the current process. The **setauthdb_r** subroutine restricts the user and group library functions to the named administrative domain for the current thread. The default behavior can be restored by using a null pointer for the value of the ***New*** parameter or an empty string for the value of the ***New*** parameter.

The string that is referenced by the ***New*** parameter must be the string `files`, `compat` or an administrative domain that is defined in the `/usr/lib/security/methods.cfg` file. The ***New*** and ***Old*** parameters are of type **authdb_t**. The **authdb_t** type is a 16-character array that contains the name of a loadable authentication module.

Note: If the `domainlessgroups` attribute is set to `true` in the `/etc/secvars.cfg` file, and if the **setauthdb** subroutine sets the administrative domain to either LDAP or `files`, the **setauthdb**

subroutine searches the user information in both the domains (LDAP and `files`) for the *group*. This `domainlessgroups` attribute behavior is restricted to the LDAP domain and the `files` domain.

Parameters

Item	Description
New	Pointer to the name of the new database module. The ***New*** parameter must reference a value module name that is contained in the `/usr/lib/security/methods.cfg` file, or one of the predefined values (`BUILTIN`, `compat`, or `files`). The empty string can be used to remove the restriction on which modules are used.
Old	Pointer to where the name of the current module is stored. A NULL value for the ***Old*** parameter can be used if the current name of the database is not wanted.

Return Values

Item	Description
0	The module search restriction is successfully changed.
-1	The module search restriction is not changed. The `errno` variable must be examined to determine the cause of the failure.

Error Codes

Item	Description
EINVAL	The `new_auth_db` parameter is longer than the permissible length of a stanza in the `/usr/lib/security/methods.cfg` file (15 characters).
ENOENT	The `new_auth_db`does not reference a valid stanza in `/usr/lib/security/methods.cfg` or one of the predefined values.

setbuf, setvbuf, setbuffer, or setlinebuf Subroutine

Purpose

Assigns buffering to a stream.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
void setbuf ( Stream,  Buffer)
FILE *Stream;
char *Buffer;

int setvbuf (Stream, Buffer, Mode, Size)
FILE *Stream;
char *Buffer;
int  Mode;
size_t  Size;
```

```
void setbuffer (Stream, Buffer, Size)
FILE *Stream;
char *Buffer;
size_t Size;
```

```
void setlinebuf (Stream)
FILE *Stream;
```

Description

The **setbuf** subroutine causes the character array pointed to by the *Buffer* parameter to be used instead of an automatically allocated buffer. Use the **setbuf** subroutine after a stream has been opened, but before it is read or written.

If the *Buffer* parameter is a null character pointer, input/output is completely unbuffered.

A constant, **BUFSIZ**, defined in the **stdio.h** file, tells how large an array is needed:

```
char buf[BUFSIZ];
```

For the **setvbuf** subroutine, the *Mode* parameter determines how the *Stream* parameter is buffered:

Item	Description
_IOFBF	Causes input/output to be fully buffered.
_IOLBF	Causes output to be line-buffered. The buffer is flushed when a new line is written, the buffer is full, or input is requested.
_IONBF	Causes input/output to be completely unbuffered.

If the *Buffer* parameter is not a null character pointer, the array it points to is used for buffering. The *Size* parameter specifies the size of the array which is used as a buffer, but all of the *Size* parameter's bytes are not necessarily used for the buffer area. Some bytes from the buffer are used for the internal buffer management. If the specified value of the *Size* parameter is less than the required value for internal buffer management, the **setvbuf** and the **setbuffer** subroutines ignore the specified buffer and performs an internal allocation of buffer.

The **BUFSIZ** constant in the **stdio.h** file is one buffer size. If the input or output is unbuffered, the **setbuf** subroutine ignores the *Buffer* and *Size* parameters. The **setbuffer** subroutine which is an alternate form of the **setbuf** subroutine, is used after the *Stream* is opened, but before it is read or written. The size of the *Buffer* character array is determined by the *Size* parameter. The *Buffer* character array is used instead of an automatically allocated buffer. If the *Buffer* parameter is a null character pointer, the input or output is completely unbuffered.

The **setbuffer** subroutine is not needed under normal circumstances because the default file I/O buffer size is optimal.

The **setlinebuf** subroutine is used to change the **stdout** or **stderr** file from block buffered or unbuffered to line-buffered. Unlike the **setbuf** and **setbuffer** subroutines, the **setlinebuf** subroutine can be used any time *Stream* is active.

A buffer is normally obtained from the **malloc** subroutine at the time of the first **getc** subroutine or **putc** subroutine on the file, except that the standard error stream, **stderr**, is normally not buffered.

Output streams directed to terminals are always either line-buffered or unbuffered.

Note: A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stream in the same block.

The **setbuffer** and **setlinebuf** subroutines are included for compatibility with Berkeley System Distribution (BSD).

Parameters

Item	Description
Stream	Specifies the input/output stream.
Buffer	Points to a character array.
Mode	Determines how the *Stream* parameter is buffered.
Size	Specifies the size of the buffer to be used.

Example

```
#include <stdio.h>

#define  SIZE  1024

int main(void)
{
    FILE *fp1;
    char buf[SIZE];
    memset( buf, '\0', sizeof( buf ));
    fp1 = fopen("file1", "r");

    /* Error Handling for fopen */

    if (setvbuf(fp1, buf, _IOFBF, SIZE) != 0)
             printf("Not proper data provided to setvbuf\n");
    if (fclose(fp1))
            perror("fclose error");
}
```

Return Values

Upon successful completion, **setvbuf** returns a value of 0. Otherwise it returns a nonzero value if a value that is not valid is given for type, or if the request cannot be honored.

setcsmap Subroutine

Purpose

Reads a code-set map file and assigns it to the standard input device.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/termios.h>
```

int setcsmap (*Path***);**
char * *Path*;

Description

The **setcsmap** subroutine reads in a code-set map file. The *path* parameter specifies the location of the code-set map file. The path is usually composed by forming a string with the **csmap** directory and the code set, as in the following example:

```
n=sprintf(path,"%s%s",CSMAP_DIR,nl_langinfo(CODESET));
```

The file is processed and according to the included informations, the **setcsmap** subroutine changes the tty configuration. Multibyte processing may be enabled, and converter modules may be pushed onto the tty stream.

Parameter

Item	Description
Path	Names the code-set map file.

Return Values

If a code set-map file is successfully opened and compiled, a value of 0 is returned. If an error occurred, a value of 1 is returned and the **errno** global variable is set to identify the error.

Error Codes

Item	Description
EINVAL	Indicates an invalid value in the code set map.
EIO	An I/O error occurred while the file system was being read.
ENOMEM	Insufficient resources are available to satisfy the request.
EFAULT	A kernel service, such as **copyin**, has failed.
ENOENT	The named file does not exist.
EACCES	The named file cannot be read.

setea Subroutine

Purpose

Sets an extended attribute value.

Syntax

```
#include <sys/ea.h>

int setea(const char *path, const char *name,
        void *value, size_t size, int flags);
int fsetea(int filedes, const char *name,
        void *value, size_t size, int flags);
int lsetea(const char *path, const char *name,
        void *value, size_t size, int flags);
```

Description

Extended attributes are name:value pairs associated with the file system objects (such as files, directories, and symlinks). They are extensions to the normal attributes that are associated with all objects in the file system (that is, the **stat(2)** data).

Do not define an extended attribute name with the 8-character prefix "(0xF8)SYSTEM(0xF8)". Prefix "(0xF8)SYSTEM(0xF8)" is reserved for system use only.

Note: 0xF8 represents a non-printable character.

The **setea** subroutine sets the value of the extended attribute identified by *name* and associated with the given *path* in the file system. The size of the value must be specified. The **fsetea** subroutine is identical to **setea**, except that it takes a file descriptor instead of a path. The **lsetea** subroutine is identical to **setea**, except, in the case of a symbolic link, the link itself is interrogated rather than the file that it refers to.

Parameters

Item	Description
path	The path name of the file.
name	The name of the extended attribute. An extended attribute name is a NULL-terminated string.
value	A pointer to the value of an attribute. The value of an extended attribute is an opaque byte stream of specified length.
size	The length of the value.
filedes	A file descriptor for the file.
flags	None are defined at this time.

Return Values

If the **setea** subroutine succeeds, 0 is returned. Upon failure, -1 is returned and **errno** is set appropriately.

Error Codes

Item	Description
EACCES	Caller lacks write permission to the base file, or lacks the appropriate ACL privileges for named attribute **write**.
EDQUOT	Because of quota enforcement, the remaining space is insufficient to store the extended attribute.
EFAULT	A bad address was passed for *path*, *name*, or *value*.
EFORMAT	File system is capable of supporting EAs, but EAs are disabled.
EINVAL	No flags should be specified.
EINVAL	A path-like name should not be used (such as **zml/file**, **.** and **..**).
ENAMETOOLONG	The *path* or *name* value is too long.
ENOSPC	The remaining space is insufficient to store the extended attribute.
ENOTSUP	Extended attributes are not supported by the file system.

The errors documented for the **stat(2)** system call are also applicable here.

setgid, setrgid, setegid, setregid, or setgidx Subroutine

Purpose

Sets the process group IDs.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
int setgid (GID)
gid_t  GID;
```

```
int setrgid (RGID)
gid_t  RGID;
```

```
int setegid (EGID)
gid_t  EGID;
```

```
int setregid (RGID, EGID)
gid_t RGID;
gid_t EGID;
```

```
#include <unistd.h>
#include <sys/id.h>

int setgidx ( which, GID )
int which;
gid_t GID;
```

Description

The **setgid**, **setrgid**, **setegid**, **setregid**, and **setgidx** subroutines set the process group IDs of the calling process. The following semantics are supported:

Item	Description
setgid	If the effective user ID of the process is the root user, the process's real, effective, and saved group IDs are set to the value of the *GID* parameter. Otherwise, the process effective group ID is reset if the *GID* parameter is equal to either the current real or saved group IDs, or one of its supplementary group IDs. Supplementary group IDs of the calling process are not changed.
setegid	The process effective group ID is reset if one of the following conditions is met: • The *EGID* parameter is equal to either the current real or saved group IDs. • The *EGID* parameter is equal to one of its supplementary group IDs. • The effective user ID of the process is the root user.
setrgid	The **EPERM** error code is always returned.
setregid	The *RGID* and *EGID* parameters can have one of the following relationships: ***RGID* != *EGID*** If the *EGID* parameter is equal to either the process's real or saved group IDs, the process effective group ID is set to the *EGID* parameter. Otherwise, the **EPERM** error code is returned. ***RGID* == *EGID*** If the effective user ID of the process is the root user, the process's real and effective group IDs are set to the *EGID* parameter. If the *EGID* parameter is equal to the process's real or saved group IDs, the process effective group ID is set to *EGID*. Otherwise, the **EPERM** error code is returned.

Item	Description
setgidx	The *which* parameter can have one of the following values: **ID_EFFECTIVE** *GID* must be either the real or saved *GID* or one of the values in the concurrent group set. The effective group ID for the current process will be set to *GID*. **ID_EFFECTIVE|ID_REAL** Invoker must have appropriate privilege. The real and effective group ID for the current process will be set to *GID*. **ID_EFFECTIVE|ID_REAL|ID_SAVED** Invoker must have appropriate privilege. The real, effective and saved group ID for the current process will be set to GID.

The **setegid**, **setrgid**, **setregid**, and **setgidx** subroutines are thread-safe.

The operating system does not support **setuid** ("setuid, setruid, seteuid, setreuid or setuidx Subroutine" on page 1918) or **setgid** shell scripts.

These subroutines are part of Base Operating System (BOS) Runtime.

Parameters

Item	Description
GID	Specifies the value of the group ID to set.
RGID	Specifies the value of the real group ID to set.
EGID	Specifies the value of the effective group ID to set.
which	Specifies which group ID values to set.

Return Values

Item	Description
0	Indicates that the subroutine was successful.
-1	Indicates the subroutine failed. The **errno** global variable is set to indicate the error.

Error Codes

If the **setgid**, **setegid**, or **setgidx** subroutine fails, one or more of the following are returned:

Item	Description
EPERM	Indicates the process does not have appropriate privileges and the *GID* or *EGID* parameter is not equal to either the real or saved group IDs of the process.
EINVAL	Indicates the value of the *GID*, *EGID* or *which* parameter is invalid.

setgroups Subroutine

Purpose

Sets the supplementary group ID of the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <grp.h>
```

int setgroups (*NumberGroups*, *GroupIDSet*)
int *NumberGroups*;
gid_t ***GroupIDSet*;

Description

The **setgroups** subroutine sets the supplementary group ID of the process. The **setgroups** subroutine cannot set more than **NGROUPS_MAX** groups in the group set. (**NGROUPS_MAX** is a constant defined in the **limits.h** file.)

Note: The routine may coredump instead of returning EFAULT when an invalid pointer is passed in case of 64-bit application calling 32-bit kernel interface.

Parameters

Item	Description
GroupIDSet	Pointer to the array of group IDs to be established.
NumberGroups	Indicates the number of entries in the *GroupIDSet* parameter.

Return Values

Upon successful completion, the **setgroups** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setgroups** subroutine fails if any of the following are true:

Item	Description
EFAULT	The *NumberGroups* and *GroupIDSet* parameters specify an array that is partially or completely outside of the process' allocated address space.
EINVAL	The *NumberGroups* parameter is greater than the **NGROUPS_MAX** value.
EPERM	A group ID in the *GroupIDSet* parameter is not presently in the supplementary group ID, and the invoker does not have root user authority.

Security

Auditing Events:

Event	Information
PROC_SetGroups	*NumberGroups*, *GroupIDSet*

setjmp or longjmp Subroutine

Purpose

Saves and restores the current execution context.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <setjmp.h>
int setjmp (Context)
jmp_buf Context;
```

```
void longjmp ( Context,  Value)
jmp_buf Context;
int Value;
```

```
int _setjmp (Context)
jmp_buf Context;
```

```
void _longjmp (Context, Value)
jmp_buf Context;
int Value;
```

Description

The **setjmp** subroutine and the **longjmp** subroutine are useful when handling errors and interrupts encountered in low-level subroutines of a program.

The **setjmp** subroutine saves the current stack context and signal mask in the buffer specified by the *Context* parameter.

The **longjmp** subroutine restores the stack context and signal mask that were saved by the **setjmp** subroutine in the corresponding *Context* buffer. After the **longjmp** subroutine runs, program execution continues as if the corresponding call to the **setjmp** subroutine had just returned the value of the *Value* parameter. The subroutine that called the **setjmp** subroutine must not have returned before the completion of the **longjmp** subroutine. The **setjmp** and **longjmp** subroutines save and restore the signal mask **sigmask (2),** while **_setjmp** and **_longjmp** manipulate only the stack context.

If a process is using the AT&T System V **sigset** interface, then the **setjmp** and **longjmp** subroutines do not save and restore the signal mask. In such a case, their actions are identical to those of the **_setjmp** and **_longjmp** subroutines.

Parameters

Item	Description
Context	Specifies an address for a **jmp_buf** structure.
Value	Indicates any integer value.

Return Values

The **setjmp** subroutine returns a value of 0, unless the return is from a call to the **longjmp** function, in which case **setjmp** returns a nonzero value.

The **longjmp** subroutine cannot return 0 to the previous context. The value 0 is reserved to indicate the actual return from the **setjmp** subroutine when first called by the program. The **longjmp** subroutine does not return from where it was called, but rather, program execution continues as if the corresponding call to **setjmp** was returned with a returned value of *Value*.

If the **longjmp** subroutine is passed a *Value* parameter of 0, then execution continues as if the corresponding call to the **setjmp** subroutine had returned a value of 1. All accessible data have values as of the time the **longjmp** subroutine is called.

Attention: If the **longjmp** subroutine is called with a *Context* parameter that was not previously set by the **setjmp** subroutine, or if the subroutine that made the corresponding call to the **setjmp** subroutine has already returned, then the results of the **longjmp** subroutine are undefined. If the **longjmp** subroutine detects such a condition, it calls the **longjmperror** routine. If **longjmperror** returns, the program is aborted. The default version of **longjmperror** prints the message: `longjmp or siglongjmp used outside of saved context` to standard error and returns. Users wishing to exit in another manner can write their own version of the **longjmperror** program.

setiopri Subroutine

Purpose

Enables the setting of a process I/O priority.

Syntax

```
short setiopri (ProcessID, IOPriority);
pid_t ProcessID;ushort IOPriority
```

Description

The `setiopri` subroutine sets the I/O scheduling priority of all threads in a process to be a constant. If the target process ID does not match the process ID of the caller, the caller must either be running as root or have an effective and real user ID that matches the target process. A smaller value for the *IOPriority* designates a higher scheduling priority. Only a few I/O devices support priorities.

Parameters

Item	Description
ProcessID	Specifies the process ID. If this value is `-1`, the current process I/O scheduling priority is set to a constant.
IOPriority	Specifies the I/O scheduling priority for the process. The *IOPriority* parameter must be in the range `IOPRIORITY_MIN`≤*IOPriority*<`IOPRIORITY_MAX`. (See the `sys/extendio.h` file.)

Return Values

Upon successful completion, the `setiopri` subroutine returns the former I/O scheduling priority of the process just changed. A returned value of `IOPRIORITY_UNSET` indicates that the I/O priority was not set. Otherwise, a value of `-1` is returned and the `errno` global variable is set to indicate the error.

Errors

Item	Description
`EINVAL`	*IOPriority* value is invalid.

Item	Description
EPERM	The calling process is not root. It does not have the same process ID as the target process, and does not have the same real effective user ID as the target process.
ESRCH	No process can be found corresponding to the specified *ProcessID*.

Implementation Specifics

1. Implementation requires an additional field in the `proc` structure.
2. The default setting for process I/O priority is `IOPRIORITY_UNSET`.
3. Once set, process I/O priorities should be inherited across a `fork`. I/O priorities should not be inherited across an `exec`.
4. The `setiopri` system call generates an auditing event using *audit_svcstart* if auditing is enabled on the system (*audit_flag* is true).

setlocale Subroutine

Purpose

Changes or queries the program's entire current locale or portions thereof.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <locale.h>
```

```
char *setlocale ( Category,  Locale)
int Category;
const char *Locale;
```

Description

The **setlocale** subroutine selects all or part of the program's locale specified by the *Category* and *Locale* parameters. The **setlocale** subroutine then changes or queries the specified portion of the locale. The **LC_ALL** value for the *Category* parameter names the entire locale (all the categories). The other *Category* values name only a portion of the program locale.

The *Locale* parameter specifies a string that provides information needed to set certain conventions in the *Category* parameter. The components of the *Locale* parameter are language and territory. Values allowed for the locale argument are the predefined **language_territory** combinations or a user-defined locale.

If a user defines a new locale, a uniquely named locale definition source file must be provided. The character collation, character classification, monetary, numeric, time, and message information should be provided in this file. The locale definition source file is converted to a binary file by the **localedef** command. The binary locale definition file is accessed in the directory specified by the **LOCPATH** environment variable.

Note: All **setuid** and **setgid** programs will ignore the **LOCPATH** environment variable.

The default locale at program startup is the C locale. A call to the **setlocale** subroutine must be made explicitly to change this default locale environment.

The locale state is common to all threads within a process.

Parameters

Item	Description
Category	Specifies a value representing all or part of the locale for a program. Depending on the value of the *Locale* parameter, these categories may be initiated by the values of environment variables with corresponding names. Valid values for the *Category* parameter, as defined in the **locale.h** file, are: **LC_ALL** Affects the behavior of a program's entire locale. **LC_COLLATE** Affects the behavior of regular expression and collation subroutines. **LC_CTYPE** Affects the behavior of regular expression, character-classification, case-conversion, and wide character subroutines. **LC_MESSAGES** Affects the content of messages and affirmative and negative responses. **LC_MONETARY** Affects the behavior of subroutines that format monetary values. **LC_NUMERIC** Affects the behavior of subroutines that format nonmonetary numeric values. **LC_TIME** Affects the behavior of time-conversion subroutines.
Locale	Points to a character string containing the required setting for the *Category* parameter. The following are special values for the *Locale* parameter: **"C"** The C locale is the locale all programs inherit at program startup. **"POSIX"** Specifies the same locale as a value of **"C"**. **""** Specifies categories be set according to locale environment variables. **NULL** Queries the current locale environment and returns the name of the locale. For more information about supported locale values for the *Locale* parameter, see Supported languages and locales in *Globalization Guide and Reference*.

Return Values

If a pointer to a string is given for the *Locale* parameter and the selection can be honored, the **setlocale** subroutine returns the string associated with the specified *Category* parameter for the new locale. If the selection cannot be honored, a null pointer is returned and the program locale is unchanged.

If a null is used for the *Locale* parameter, the **setlocale** subroutine returns the string associated with the *Category* parameter for the program's current locale. The program's locale is not changed.

A subsequent call with the string returned by the **setlocale** subroutine, and its associated category, will restore that part of the program locale. The string returned is not modified by the program, but can be overwritten by a subsequent call to the **setlocale** subroutine.

setosuuid Subroutine

Purpose

Sets the operating system Universal Unique Identifier (UUID).

Library

Standard C Library (**libc.a**)

Syntax

```
#include <uuid.h>
int setosuuid (uuid)
uuid_t * uuid;
```

Description

The **setosuuid** subroutine saves the UUID pointed to by the *uuid* parameter as the operating system UUID in the AIX kernel. This subroutine can only be run with the root privileges.

Note:

The UUID of the AIX operating system can be reset to a new system generated UUID using the **chdev** command. Setting the UUID to an empty string will cause the system to generate a new UUID:

```
chdev -l sys0 -a os_uuid=""
```

The UUID of the AIX operating system can be reset to a specific UUID using the **chdev** command:

```
chdev -l sys0 -a os_uuid="<uuid_string>"
```

If the **chdev** command is used to reset the UUID to an invalid UUID, the system will disregard this UUID and generate a new one.

Parameters

Item	Description
uuid	Specifies the UUID to be saved as the operating system UUID.

Return Values

Upon successful completion the **setosuuid** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EPERM	The process does not have the appropriate privileges.
EFAULT	The address in parameter *uuid* is invalid.

setpagvalue or setpagvalue64 Subroutine

Purpose

Sets the Process Authentication Group (PAG) value for a given PAG type.

Library

Security Library (`libc.a`)

Syntax

```
#include <pag.h>

int setpagvalue ( name, value )
char * name;
int  value;

uint64_t setpagvalue64( name, value );
char * name;
uint64  value;
```

Description

The `setpagvalue` or `setpagvalue64` subroutine sets the PAG value for a given PAG name. For these functions to succeed, the PAG name must be registered with the operating system before these subroutines are called.

Parameters

Item	Description
name	A 1-character to 4-character, NULL-terminated name for the PAG type. Typical values include `afs`, `dfs`, `pki`, and `krb5`.
value	New PAG value for the given *name*.

Return Values

The `setpagvalue` and `setpagvalue64` subroutines return a PAG value upon successful completion. Upon a failure, a value of `-1` is returned and the `errno` global variable is set to indicate the error.

Error Codes

The `setpagvalue` and `setpagvalue64` subroutines fail if the following condition is true:

Item	Description
`EINVAL`	The named PAG type does not exist as part of the table.

Other errors might be set by subroutines invoked by the `setpagvalue` and `setpagvalue64` subroutines.

setpcred Subroutine

Purpose

Sets the current process credentials.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int setpcred ( User,  Credentials)
char **Credentials;
char *User;
```

Description

The **setpcred** subroutine sets a process' credentials according to the *Credentials* parameter. If the *User* parameter is specified, the credentials defined for the user in the user database are used. If the *Credentials* parameter is specified, the credentials in this string are used. If both the *User* and *Credentials* parameters are specified, both the user's and the supplied credentials are used. However, the supplied credentials of the *Credentials* parameter will override those of the user. At least one parameter must be specified.

The **setpcred** subroutine requires the **setpenv** subroutine to follow it.

Note: If the **auditwrite** subroutine is to be called from a program invoked from the **inittab** file, the **setpcred** subroutine should be called first to establish the process' credentials.

Item	Description
User	Specifies the user for whom credentials are being established.
Credentials	Defines specific credentials to be established. This parameter points to an array of null-terminated character strings that may contain the following values. The last character string must be null. **LOGIN_USER=%s** Login user name **REAL_USER=%s** Real user name **REAL_GROUP=%s** Real group name **GROUPS=%s** Supplementary group ID **AUDIT_CLASSES=%s** Audit classes **RLIMIT_CPU=%d** Process soft CPU limit **RLIMIT_FSIZE=%d** Process soft file size **RLIMIT_DATA=%d** Process soft data segment size **RLIMIT_STACK=%d** Process soft stack segment size **RLIMIT_CORE=%d** Process soft core file size **RLIMIT_RSS=%d** Process soft resident set size **RLIMIT_CORE_HARD=%d** Process hard core file size **RLIMIT_CPU_HARD=%d** Process hard CPU limit

Item	Description
	RLIMIT_DATA_HARD=%d Process hard data segment size **RLIMIT_FSIZE_HARD=%d** Process hard file size **RLIMIT_RSS_HARD=%d** Process hard resident set size **RLIMIT_STACK_HARD=%d** Process hard stack segment size **UMASK=%o** Process **umask** (file creation mask) **ROLES=%s** Role names **DOMAINS=%s** Domain names A process must have root user authority to set all credentials except the UMASK credential. **Resource Hard Soft** RLIMIT_CORE unlimited %d RLIMIT_CPU %d %d RLIMIT_DATA unlimited %d RLIMIT_FSIZE %d %d RLIMIT_RSS unlimited %d RLIMIT_STACK unlimited %d

The soft limit credentials will override the equivalent hard limit credentials that may proceed them. To set the hard limits, the hard limit credentials should follow the soft limit credentials.

Note: The resident set size (RSS) hard limit credentials and RSS soft limit credentials are not implemented by the system.

Return Values

Upon successful return, the **setpcred** subroutine returns a value of 0. If **setpcred** fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setpcred** subroutine fails if one or more of the following are true:

Item	Description
EINVAL	The *Credentials* parameter contains invalid credentials specifications.
EINVAL	The *User* parameter is null and the *Credentials* parameter is either null or points to an empty string.
EPERM	The process does not have the proper authority to set the requested credentials.

Other errors may be set by subroutines invoked by the **setpcred** subroutine.

setpenv Subroutine

Purpose

Sets the current process environment.

Library

Security Library (**libc.a**)

Syntax

#include <usersec.h>

int setpenv (*User*, *Mode*, *Environment*, *Command***) char ***User**; int** *Mode***; char ****Environment**;**
char ***Command;**

Description

The **setpenv** subroutine first sets the environment of the current process according to its parameter values, and then sets the working directory and runs a specified command. If the *User* parameter is specified, the process environment is set to that of the specified user, the user's working directory is set, and the specified command run. If the *User* parameter is not specified, then the environment and working directory are set to that of the current process, and the command is run from this process. The environment consists of both user-state and system-state environment variables.

Note: The **setpenv** subroutine requires the **setpcred** subroutine to precede it.

The **setpenv** subroutine performs the following steps:

Item	Description
Setting the Process Environment	The first step involves changing the user-state and system-state environment. Since this is dependent on the values of the *Mode* and *Environment* parameters, see the description for the *Mode* parameter for more information.
Setting the Process Current Working Directory	After the user-state and system-state environment is set, the working directory of the process may be set. If the *Mode* parameter includes the **PENV_INIT** value, the current working directory is changed to the user's initial login directory (defined in the **/etc/passwd** file). Otherwise, the current working directory is unchanged.

Item	Description
Executing the Initial Program	After the working directory of the process is reset, the initial program (usually the shell interpreter) is executed. If the *Command* parameter is null, the shell from the user database is used. If the parameter is not defined, the shell from the user-state environment is used and the *Command* parameter defaults to the **/usr/bin/sh** file. If the *Command* parameter is not null, it specifies the command to be executed. If the *Mode* parameter contains the **PENV_ARGV** value, the *Command* parameter is assumed to be in the **argv** structure and is passed to the **execve** subroutine. The string contained in the *Command* parameter is used as the *Path* parameter of the **execve** subroutine. If the *Mode* parameter does not contain **PENV_ARGV** value, the *Command* parameter is parsed into an **argv** structure and executed. If the *Command* parameter contains the **$SHELL** value, substitution is done prior to execution. **Note:** This step will fail if the *Command* parameter contains the **$SHELL** value but the user-state environment does not contain the **SHELL** value.

Parameters

Command
Specifies the command to be executed. If the *Mode* parameter contains the **PENV_ARGV** value, then the *Command* parameter is assumed to be a valid argument vector for the **execv** subroutine.

Environment
Specifies the value of user-state and system-state environment variables in the same format returned by the **getpenv** subroutine. The user-state variables are prefaced by the keyword **USRENVIRON:**, and the system-state variables are prefaced by the keyword **SYSENVIRON:**. Each variable is defined by a string of the form **var**=*value*, which is an array of null-terminated character pointers.

Mode
Specifies how the **setpenv** subroutine is to set the environment and run the command. This parameter is a bit mask and must contain only one of the following values, which are defined in the **usersec.h** file:

PENV_INIT
The user-state environment is initialized as follows:

AUTHSTATE
Retained from the current environment. If the **AUTHSTATE** value is not present, it is defaulted to the **compat** value.

KRB5CCNAME
Retained from the current environment. This value is defined if you authenticated through the Distributed Computing Environment (DCE).

USER
Set to the name specified by the *User* parameter or to the name corresponding to the current real user ID. The name is shortened to a maximum of **PW_USERNAME_LEN**, including the trailing NUL character. **PW_USERNAME_LEN** is the running system's maximum value. The value of **PW_USERNAME_LEN** can be at the most MAXIMPL_LOGIN_NAME_MAX (or 256 characters), and must be at least 9 characters.

LOGIN
Set to the name specified by the *User* parameter or to the name corresponding to the current real user ID. If set by the *User* parameter, this value is the complete login name, which may include a DCE cell name.

LOGNAME
Set to the current system environment variable **LOGNAME**.

TERM
Retained from the current environment. If the **TERM** value is not present, it is defaulted to an **IBM6155**.

SHELL
Set from the initial program defined for the real user ID of the current process. If no program is defined, then the **/usr/bin/sh** shell is used as the default.

HOME
Set from the home directory defined for the real user ID of the current process. If no home directory is defined, the default is **/home/guest**.

PATH
Set initially to the value for the **PATH** value in the **/etc/environment** file. If not set, it is destructively replaced by the default value of **PATH=/usr/bin:$HOME:.** (The final period specifies the working directory). The **PATH** variable is destructively replaced by the **usrenv** attribute for this user in the **/etc/security/environ** file if the **PATH** value exists in the /**etc**/**environment** file.

The following files are read for additional environment variables:

/etc/environment
Variables defined in this file are added to the environment.

/etc/security/environ
Environment variables defined for the user in this file are added to the user-state environment.

The user-state variables in the *Environment* parameter are added to the user-state environment. These are preceded by the **USRENVIRON:** keyword.

The system-state environment is initialized as follows:

LOGNAME
Set to the current **LOGNAME** value in the protected user environment. The **login** (**tsm**) command passes this value to the **setpenv** subroutine to ensure correctness.

NAME
Set to the login name corresponding to the real user ID.

TTY
Set to the TTY name corresponding to standard input.

The following file is read for additional environment variables:

/etc/security/environ
The system-state environment variables defined for the user in this file are added to the environment. The system-state variables in the *Environment* parameter are added to the environment. These are preceded by the **SYSENVIRON** keyword.

PENV_DELTA
The existing user-state and system-state environment variables are preserved and the variables defined in the *Environment* parameter are added.

PENV_RESET
The existing environment is cleared and totally replaced by the content of the *Environment* parameter.

PENV_KLEEN
Closes all open file descriptors, except 0, 1, and 2, before executing the command. This value must be logically ORed with **PENV_DELTA**, **PENV_RESET**, or **PENV_INIT**. It cannot be used alone.

PENV_NOPROF
The new shell will not be treated as a login shell. Only valid when used with the **PENV_INIT** flag.

For both system-state and user-state environments, variable substitution is performed.

The *Mode* parameter may also contain:

Item	Description
PENV_ARGV	Specifies that the *Command* parameter is already in **argv** format and need not be parsed. This value must be logically ORed with **PENV_DELTA**, **PENV_RESET**, or **PENV_INIT**. It cannot be used alone.

Item	Description
User	Specifies the user name whose environment and working directory is to be set and the specified command run. If a null pointer is given, the current real uid is used to determine the name of the user.

Return Values

If the environment was successfully established, this function does not return. If the **setpenv** subroutine fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setpenv** subroutine fails if one or more of the following are true:

Item	Description
EINVAL	The *Mode* parameter contains values other than **PENV_INIT**, **PENV_DELTA**, **PENV_RESET**, or **PENV_ARGV**.
EINVAL	The *Mode* parameter contains more than one of **PENV_INIT**, **PENV_DELTA**, or **PENV_RESET** values.
EINVAL	The *Environment* parameter is neither null nor empty, and does not contain a valid environment string.

Item	Description
EPERM	The caller does not have read access to the environment defined for the system, or the user does not have permission to change the specified attributes.

Other errors may be set by subroutines invoked by the **setpenv** subroutine.

setpgid or setpgrp Subroutine

Purpose

Sets the process group ID.

Libraries

setpgid: Standard C Library (**libc.a**)

setpgrp: Standard C Library (**libc.a**);

Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <unistd.h>
```

pid_t setpgid (*ProcessID*, *ProcessGroupID*)
pid_t *ProcessID*, *ProcessGroupID*;

```
pid_t setpgrp ( )
```

Description

The **setpgid** subroutine is used either to join an existing process group or to create a new process group within the session of the calling process. The process group ID of a session leader does not change. Upon return, the process group ID of the process having a process ID that matches the *ProcessID* value is set to the *ProcessGroupID* value. As a special case, if the *ProcessID* value is 0, the process ID of the calling process is used. If *ProcessGroupID* value is 0, the process ID of the indicated process is used.

This function is implemented to support job control.

The **setpgrp** subroutine in the **libc.a** library supports a subset of the function of the **setpgid** subroutine. It has no parameters. It sets the process group ID of the calling process to be the same as its process ID and returns the new value.

In BSD systems, the **setpgrp** subroutine is defined with two parameters, as follows:

```
pid_t setpgrp (ProcessID, ProcessGroup)
pid_t ProcessID, ProcessGroup;
```

Parameters

Item	Description
ProcessID	Specifies the process whose process group ID is to be changed.
ProcessGroupID	Specifies the new value of calling process group ID.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setpgid** subroutine is unsuccessful if one or more of the following is true:

Item	Description
EACCES	The value of the *ProcessID* parameter matches the process ID of a child process of the calling process and the child process has successfully executed one of the **exec** subroutines.
EINVAL	The value of the *ProcessGroupID* parameter is less than 0, or is not a valid value.
ENOSYS	The **setpgid** subroutine is not supported by this implementation.
EPERM	The process indicated by the value of the *ProcessID* parameter is a session leader.

Item	Description
EPERM	The value of the *ProcessID* parameter matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process.
EPERM	The value of the *ProcessGroupID* parameter is valid, but does not match the process ID of the process indicated by the *ProcessID* parameter. There is no process with a process group ID that matches the value of the *ProcessGroupID* parameter in the same session as the calling process.
ESRCH	The value of the *ProcessID* parameter does not match the process ID of the calling process of a child process of the calling process.

setppdmode Subroutine

Purpose

Sets the access mode of partitioned directories.

Syntax

```
#include <sys/secconf.h>
```

```
int setppdmode(Mode)
int Mode;
```

Description

The **setppdmode** subroutine sets the access mode of partitioned directories.

Parameters

Item	Description
Mode	Specifies the access mode of partitioned directories. The *Mode* parameter can be one of the following values: **PD_REAL** Sets the access mode to the real mode. **PD_VIRTUAL** Sets the access mode to the virtual mode.

Return Values

Item	Description
0	Successful
≠0	Unsuccessful

setppriv Subroutine

Purpose

Sets the privilege sets associated with a process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/priv.h>

int setppriv(pid, effective, maximum, inheritable,limiting)
pid_t pid;
privg_t * effective, maximum, inheritable, limiting;
```

Description

The **setppriv** subroutine sets the effective (EPS), maximum (MPS), inheritable (IPS) and limiting (LPS) privilege sets for the process as specified by the *pid* parameter. If the value of the *pid* parameter is negative, the privileges of the calling process are modified. The PV_PROC_PRIV privilege is needed in the effective set when a process wants to change the maximum or inheritable privilege set of any process or the effective privilege sets of another process. The calling process does not require a privilege to reduce its own maximum or inheritable privilege set or to modify its own effective privilege set. The limiting privilege acts as a ceiling for the maximum and inheritable privilege. The maximum privilege acts as a ceiling for the effective privilege. The effective privilege is the current privilege of the process per the *pid* parameter.

If the effective, maximum, inheritable or limiting privilege set has a value of null, the corresponding privilege set of the process remains unchanged. At least one of the effective, maximum, inheritable and limiting privilege sets must not have a value of null.

When the privilege of the process identified by the *pid* parameter is modified, the privilege sets of the process have the following proper relationship: the new effective privilege set of the process must be a subset of the new maximum privilege set of the process. Otherwise, the call fails.

Parameters

Item	Description
pid	Indicates that the process for which the privilege set change is requested.
effective	Sets the effective privilege set, which is used to override system restrictions.
maximum	Sets the maximum privilege set over which a process has control.
inheritable	Sets the inheritable privilege set, which is passed to the EPS and MPS of a child process.
limiting	Sets the limiting privilege set, which is the maximum possible privilege set that the process can have.

Return Values

Item	Description
0	The subroutine ran successfully.
-1	An error occurred. The **errno** global variable is set to indicate the error.

Error Codes

The **setppriv** subroutine fails if any of the following are true:

Item	Description
EFAULT	The effective, maximum, inheritable or limiting privilege set is an illegal address.
EINVAL	The value of the effective, maximum, inheritable, and limiting privilege set passed are all null.
EPERM	The calling process does not have the PV_PROC_PRIV or MAC write privilege (in Trusted AIX) to modify a process privilege set.
ESRCH	No process has an ID equal to the value specified by the *pid* parameter.

setpri Subroutine

Purpose

Sets a process scheduling priority to a constant value.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/sched.h>

int setpri ( ProcessID,  Priority)
pid_t ProcessID;
int Priority;
```

Description

The **setpri** subroutine sets the scheduling priority of all threads in a process to be a constant. All threads have their scheduling policies changed to **SCHED_RR**. A process nice value and CPU usage can no longer be used to determine a process scheduling priority. Only processes that have root user authority can set a process scheduling priority to a constant.

Parameters

Item	Description
ProcessID	Specifies the process ID. If this value is 0 then the current process scheduling priority is set to a constant.
Priority	Specifies the scheduling priority for the process. A lower number value designates a higher scheduling priority. The *Priority* parameter must be in the range **PRIORITY_MIN** <= *Priority* < **PRIORITY_MAX**. (See the **sys/sched.h** file.)

Return Values

Upon successful completion, the **setpri** subroutine returns the former scheduling priority of the process just changed. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setpri** subroutine is unsuccessful if one or more of the following is true:

Item	Description
EINVAL	The priority specified by the *Priority* parameter is outside the range of acceptable priorities.
EPERM	The process executing the **setpri** subroutine call does not have root user authority.
ESRCH	No process can be found corresponding to that specified by the *ProcessID* parameter.

setpwdb or endpwdb Subroutine

Purpose

Opens or closes the authentication database.

Library

Security Library (**libc.a**)

Syntax

```
#include <userpw.h>
```

int setpwdb (*Mode*)
int *Mode*;

```
int endpwdb ( )
```

Description

These functions are used to open and close access to the authentication database. Programs that call either the **getuserpw** or **putuserpw** subroutine should call the **setpwdb** subroutine to open the database and the **endpwdb** subroutine to close the database.

The **setpwdb** subroutine opens the authentication database in the specified mode, if it is not already open. The open count is increased by 1.

The **endpwdb** subroutine decreases the open count by one and closes the authentication database when this count drops to 0. Subsequent references to individual data items can cause a memory access violation. The **endpwdb** subroutine also frees the space that was allocated by either the **getuserpw, putuserpw,** or **putuserpwhist** subroutine. For security reasons, freeing the space clears the password field. Any uncommitted changed data is lost.

Parameters

Item	Description
Mode	Specifies the mode of the open. This parameter may contain one or more of the following values, defined in the **usersec.h** file: **S_READ** Specifies read access. **S_WRITE** Specifies update access.

Return Values

The **setpwdb** and **endpwdb** subroutines return a value of 0 to indicate success. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setpwdb** and **endpwdb** subroutines fail if the following is true:

Item	Description
EACCES	Access permission is denied for the data request.

Both of these functions return errors from other subroutines.

Security

Access Control: The calling process must have access to the authentication data.

Files Accessed:

Modes	File
rw	/etc/security/passwd
rw	/etc/passwd

setroledb or endroledb Subroutine

Purpose

Opens and closes the role database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

```
int setroledb(Mode)
int Mode;
```

```
int endroledb
```

Description

These functions may be used to open and close access to the role database. Programs that call the **getroleattr** subroutine should call the **setroledb** subroutine to open the role database and the **endroledb** subroutine to close the role database.

The **setroledb** subroutine opens the role database in the specified mode, if it is not already open. The open count is increased by 1.

The **endroledb** subroutine decreases the open count by 1 and closes the role database when this count goes to 0. Any uncommitted changed data is lost.

Parameters

Item	Description
Mode	Specifies the mode of the open. This parameter may contain one or more of the following values defined in the **usersec.h** file: **S_READ** Specifies read access. **S_WRITE** Specifies update access.

Return Values

The **setroledb** and **endroledb** subroutines return a value of 0 to indicate success. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setroledb** subroutine fails if the following is true:

Item	Description
EACCES	Access permission is denied for the data request.

Both subroutines return errors from other subroutines.

Security

Files Accessed: The calling process must have access to the role data.

Mode File **rw/etc/security/roles**

setroles Subroutine

Purpose

Set the role IDs of the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/cred.h>

int setroles(roles, nroles)
rid_t *roles;
int nroles;
```

Description

The **setroles** subroutine sets the supplementary role ID of the process. The number of roles that the **setroles** subroutine can set is no greater than the value specified by the MAX_ROLES constant in the **cred** structure of a process. The MAX_ROLES constant is defined in the **sys/cred.h** header file.

Parameters

Item	Description
roles	Points to the array of role IDs to be established.
nroles	Indicates the number of entries in the roles parameter.

Return Values

Item	Description
0	The subroutine ran successfully.
-1	An error occurred. The **errno** global variable is set to indicate the error.

Error Codes

The **setroles** subroutine fails if any of the following are true:

Item	Description
EFAULT	The *roles* and *nroles* parameters specify an array that is partially or completely outside of the process' allocated address space.
EINVAL	The value of the *nroles* parameter is either less than 0 or greater than the MAX_ROLES value.
EPERM	The calling process does not have the PV_DAC_RID privilege in its effective privilege set.

setsecorder Subroutine

Purpose

Sets the order of domains for certain security databases.

Library

Standard C Library (**libc.a**)

Syntax

```
int setsecorder (name, value)
    char *name;
    char *value;
```

Description

The **setsecorder** subroutine sets the value of the domain order to the *value* parameter for the name database. The new domain order overrides the setting from any previous **setsecorder** call, and the setting specified in the **/etc/sncontrol.conf** file. A null value pointer or a null value resets the setting made by a previous **setsecorder** call, forcing the concerned library subroutines to follow the value defined in **/etc/sncontrol.conf** file.

Parameters

Item	Description
name	Specifies the database name whose domain order is to be set. Valid values and the affected library subroutines are as follows: **authorizations** The **getauthattr**, **getauthattrs**, **putauthattr**, and **putauthattrs** subroutines. **roles** The **getroleattr**, **getroleattrs**, **putroleattr**, and **putroleattrs** subroutines. **privcmds** The **getcmdattr**, **getcmdattrs**, **putcmdattr**, and **putcmdattrs** subroutines. **privdevs** The **getdevattr**, **getdevattrs**, **putdevattr**, and **putdevatttrs** subroutines. **privfiles** The **gettrviattr**, **gettrviattrs**, **putdevattr**, and **putdevattrs** subroutines.
value	Specifies a comma-separated list of modules. The following values are valid: **files** Specifies the local module. **LDAP** Specifies the LDAP module. The system must be configured as an LDAP client to use this setting.

Return Values

Item	Description
0	The domain order has been set successfully.
-1	The domain order cannot be set. The **errno** variable is set to indicate the failure.

Error Codes

The **setsecorder** subroutine fails if one of the following codes is true.

Item	Description
EINVAL	The *name* parameter refers to an unsupported database.
EINVAL	The *value* parameter contains module names that do not refer to a valid stanza in the **/usr/lib/security/methods.cfg** file or one of the predefined values.
ENOMEM	Unable to allocate memory.

setsid Subroutine

Purpose

Creates a session and sets the process group ID.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
pid_t setsid (void)
```

Description

The **setsid** subroutine creates a new session if the calling process is not a process group leader. Upon return, the calling process is the session leader of this new session, the process group leader of a new process group, and has no controlling terminal. The process group ID of the calling process is set equal to its process ID. The calling process is the only process in the new process group and the only process in the new session.

Return Values

Upon successful completion, the value of the new process group ID is returned. Otherwise, (**pid_t**) -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setsid** subroutine is unsuccessful if the following is true:

Item	Description
EPERM	The calling process is already a process group leader, or the process group ID of a process other than the calling process matches the process ID of the calling process.

setscrreg or wsetscrreg Subroutine

Purpose

Creates a software scrolling region within a window.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
setscrreg( Tmargin,  Bmargin)
int Tmargin, Bmargin;
```

```
wsetscrreg( Window, Tmargin, Bmargin)
WINDOW *Window;
int Tmargin, Bmargin;
```

Description

The **setscrreg** and **wsetscrreg** subroutines create a software scrolling region within a window. Use the **setscrreg** subroutine with the stdscr and the the **wsetscrreg** subroutine with user-defined windows.

You pass the **setscrreg** subroutines values for the top line and bottom line of the region. If the **setscrreg** subroutine and **scrollok** subroutine are enabled for the region, any attempt to move off the line specified by the *Bmargin* parameter causes all the lines in the region to scroll up one line.

Note: Unlike the **idlok** subroutine, the **setscrreg** subroutines have nothing to do with the use of a physical scrolling region capability that the terminal may or may not have.

Parameters

Item	Description
Bmargin	Specifies the last line number in the scrolling region.
Tmargin	Specifies the first line number in the scrolling region (0 is the top line of the window.)
Window	Specifies the window to place the scrolling region in. You specify this parameter only with the **wsetscrreg** subroutine.

Examples

1. To set a scrolling region starting at the 10th line and ending at the 30th line in the stdscr, enter:

   ```
   setscrreg(9, 29);
   ```

   Note: Zero is always the first line.
2. To set a scrolling region starting at the 10th line and ending at the 30th line in the user-defined window `my_window`, enter:

   ```
   WINDOW *my_window;
   wsetscrreg(my_window, 9, 29);
   ```

setsyx Subroutine

Purpose

Sets the coordinates of the virtual screen cursor.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

setsyx(*Y*, *X***)**
int *Y*, *X*;

Description

The **setsyx** subroutine sets the coordinates of the virtual screen cursor to the specified row and column coordinates. If *Y* and *X* are both -1, then the **leaveok** flag is set. (**leaveok** may be set by applications that do not use the cursor.)

The **setsyx** subroutine is intended for use in combination with the **getsyx** subroutine. These subroutines should be used by a user-defined function that manipulates curses windows but wants the position of the cursor to remain the same. Such a function would do the following:

- Call the **getsyx** subroutine to obtain the current virtual cursor coordinates.
- Continue processing the windows.
- Call the **wnoutrefresh** subroutine on each window manipulated.
- Call the **setsyx** subroutine to reset the current virtual cursor coordinates to the original values.

- Refresh the display by calling the **doupdate** subroutine.

Parameters

Item	Description
X	Specifies the column to set the virtual screen cursor to.
Y	Specifies the row to set the virtual screen cursor to.

setuid, setruid, seteuid, setreuid or setuidx Subroutine

Purpose

Sets the process user IDs.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
int setuid (UID)
uid_t  UID;
```

```
int setruid (RUID)
uid_t  RUID;
```

```
int seteuid (EUID)
uid_t  EUID;
```

```
int setreuid (RUID, EUID)
uid_t RUID;
uid_t EUID;
```

```
#include <unistd.h>
#include <sys/id.h>
```

```
int setuidx (which, UID)
int which;
uid_t UID;
```

Description

The **setuid**, **setruid**, **seteuid**, and **setreuid** subroutines reset the process user IDs. The following semantics are supported:

Item	Description
setuid	If the effective user ID of the process is the root user, the process's real, effective, and saved user IDs are set to the value of the *UID* parameter. Otherwise, the process effective user ID is reset if the *UID* parameter specifies either the current real or saved user IDs.
seteuid	The process effective user ID is reset if the *UID* parameter is equal to either the current real or saved user IDs or if the effective user ID of the process is the root user.
setruid	The **EPERM** error code is always returned. Processes cannot reset only their real user IDs.

Item	Description
setreuid	The *RUID* and *EUID* parameters can have the following two possibilities: ***RUID* != *EUID*** If the *EUID* parameter specifies either the process's real or saved user IDs, the process effective user ID is set to the *EUID* parameter. Otherwise, the **EPERM** error code is returned. ***RUID*== *EUID*** If the process effective user ID is the root user, the process's real and effective user IDs are set to the *EUID* parameter. Otherwise, the **EPERM** error code is returned. If both the real user ID and effective user ID are changed, the saved user ID is set to the new effective user ID. Note that this change results in a loss of original privileges.
setuidx	The **setuidx** subroutine does not modify the privileges of the process after the user ID of the process has been modified. To modify the privileges and the user ID of a process, use the **setpriv** subroutine and the **setuidx** subroutine together. The *which* parameter can have one of the following values: **ID_EFFECTIVE** *UID* must be either the real or saved user ID. The effective user ID for the current process will be set to *UID*. **ID_EFFECTIVE|ID_REAL** Invoker must have appropriate privilege. The real and effective user ID for the current process will be set to *UID*. **ID_EFFECTIVE|ID_REAL|ID_SAVED** Invoker must have appropriate privilege. The real, effective and saved user ID for the current process will be set to *UID*. **ID_LOGIN** Invoker must have appropriate privilege. The login user ID for the current process will be set to *UID*.

The real and effective user ID parameters can have a value of -1. If the value is -1, the actual value for the *UID* parameter is set to the corresponding current the *UID* parameter of the process.

The operating system does not support **setuid** or **setgid** (“setgid, setrgid, setegid, setregid, or setgidx Subroutine” on page 1891) shell scripts.

These subroutines are part of Base Operating System (BOS) Runtime.

Parameters

Item	Description
UID	Specifies the user ID to set.
EUID	Specifies the effective user ID to set.
RUID	Specifies the real user ID to set.
which	Specifies which user ID values to set.

Return Values

Upon successful completion, the **setuid**, **seteuid**, **setreuid**, and **setuidx** subroutines return a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setuid**, **seteuid**, **setreuid**, and **setuidx** subroutines are unsuccessful if either of the following is true:

Item	Description
EINVAL	The value of the *UID* or *EUID* parameter is not valid.
EPERM	The process does not have the appropriate privileges and the *UID* and *EUID* parameters are not equal to either the real or saved user IDs of the process.

Examples

The following example shows using the **setuidx** and **setpriv** subroutines together:

```
#include <sys/id.h>
#include <sys/priv.h>

int main(void) {

   int uid=206;
   priv_t  priv;

   bzero(priv.pv_priv, sizeof(priv.pv_priv));

   if (setuidx(ID_EFFECTIVE|ID_REAL|ID_SAVED|ID_LOGIN,uid) < 0) {
      perror("setuidx error");
      exit(errno);
   }

   if(setpriv(PRIV_SET|PRIV_INHERITED|PRIV_EFFECTIVE|PRIV_BEQUEATH,&priv,sizeof(priv_t))<0) {
      perror("setpriv error");
      exit(errno);
   }

   exit (0);
}
```

setuserdb or enduserdb Subroutine

Purpose

Opens and closes the user database.

Library

Security Library (**libc.a**)

Syntax

```
#include <usersec.h>
```

int setuserdb (*Mode*)
int *Mode*;

```
int enduserdb ( )
```

Description

These functions may be used to open and close access to the user database. Programs that call either the **getuserattr** or **getgroupattr** subroutine should call the **setuserdb** subroutine to open the user database and the **enduserdb** subroutine to close the user database.

The **setuserdb** subroutine opens the user database in the specified mode, if it is not already open. The open count is increased by 1.

The **enduserdb** subroutine decreases the open count by 1 and closes the user database when this count goes to 0. Any uncommitted changed data is lost.

Parameters

Item	Description
Mode	Specifies the mode of the open. This parameter may contain one or more of the following values defined in the **usersec.h** file: **S_READ** Specifies read access **S_WRITE** Specifies update access.

Return Values

The **setuserdb** and **enduserdb** subroutines return a value of 0 to indicate success. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **setuserdb** subroutine fails if the following is true:

Item	Description
EACCES	Access permission is denied for the data request.

Both subroutines return errors from other subroutines.

Security

Files Accessed: The calling process must have access to the user data. Depending on the actual attributes accessed, this may include:

Item	Description
Modes	File
rw	/etc/passwd
rw	/etc/group
rw	/etc/security/user
rw	/etc/security/limits
rw	/etc/security/group
rw	/etc/security/environ

setupterm Subroutine

Purpose

Initializes the terminal structure with the values in the **terminfo** database.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
#include <term.h>
```

setupterm(*Term*, *FileNumber*, *ErrorCode***)**
char ***Term*;**
int *FileNumber***;**
int ***ErrorCode*;**

Description

The **setupterm** subroutine determines the number of lines and columns available on the output terminal. The **setupterm** subroutine calls the **termdef** subroutine to define the number of lines and columns on the display. If the **termdef** subroutine cannot supply this information, the **setupterm** subroutine uses the values in the **terminfo** database.

The **setupterm** subroutine initializes the terminal structure with the terminal-dependent capabilities from **terminfo**. This routine is automatically called by the **initscr** and **newterm** subroutines. The **setupterm** subroutine deals directly with the **terminfo** database.

Two of the terminal-dependent capabilities are the lines and columns. The **setupterm** subroutine populates the lines and column fields in the terminal structure in the following manner:

1. If the environment variables **LINES** and **COLUMNS** are set, the **setupterm** subroutine uses these values.
2. If the environment variables are not set, the **setupterm** subroutine obtains the lines and columns information from the tty subsystem.
3. As a last resort, the **setupterm** subroutine uses the values defined in the **terminfo** database.

   Note: These may or may not be the same as the values in the **terminfo** database.

The simplest call is **setupterm((char*) 0, 1, (int*) 0)**, which uses all defaults.

After the call to the **setupterm** subroutine, the **cur_term** global variable is set to point to the current structure of terminal capabilities. A program can use more than one terminal at a time by calling the **setupterm** subroutine for each terminal and then saving and restoring the **cur_term** variable.

Parameters

Item	Description
ErrorCode	Specifies a pointer to an integer to return the error code to. If a null pointer (0) is passed for this parameter, no status is returned. An error causes the **setupterm** subroutine to print an error message and exit instead of returning.
FileNumber	Specifies the output files file descriptor (1 equals standard output).
Term	Specifies the terminal name. If 0 is passed for this parameter, the value of the **$TERM** environment variable is used.

Return Values

One of the following status values is stored into the integer pointed to by the *ErrorCode* parameter:

Item	Description
1	Successful completion.
0	No such terminal.
-1	An error occurred while locating the **terminfo** database.

Example

To determine the current terminal's capabilities using **$TERM** as the terminal name, standard output as output, and returning no error codes, enter:

```
setupterm((char*) 0, 1, (int*) 0);
```

sgetl or sputl Subroutine

Purpose

Accesses long numeric data in a machine-independent fashion.

Library

Object File Access Routine Library (**libld.a**)

Syntax

```
long sgetl ( Buffer)
char *Buffer;

void sputl (Value, Buffer)
long  Value;
char *Buffer;
```

Description

The **sgetl** subroutine retrieves four bytes from memory starting at the location pointed to by the *Buffer* parameter. It then returns the bytes as a long *Value* with the byte ordering of the host machine.

The **sputl** subroutine stores the four bytes of the *Value* parameter into memory starting at the location pointed to by the *Buffer* parameter. The order of the bytes is the same across all machines.

Using the **sputl** and **sgetl** subroutines together provides a machine-independent way of storing long numeric data in an ASCII file. For example, the numeric data stored in the portable archive file format can be accessed with the **sputl** and **sgetl** subroutines.

Parameters

Item	Description
Value	Specifies a 4-byte value to store into memory.
Buffer	Points to a location in memory.

shm_open Subroutine

Purpose

Opens a shared memory object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>

int shm_open (name, oflag, mode)
const char *name;
int oflag;
mode_t mode;
```

Description

The **shm_open** subroutine establishes a connection between a shared memory object and a file descriptor. It creates an open file description that refers to the shared memory object and a file descriptor that refers to that open file description. This file descriptor is used by other subroutines to refer to that shared memory object.

The *name* parameter points to a string naming a shared memory object. The *name* parameter does not appear in the file system and is not visible to other subroutines that take pathnames as arguments.The *name* parameter must conform to the construction rules for a pathname.

If successful, the **shm_open** subroutine returns a file descriptor for the shared memory object that is the lowest numbered file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other processes. The **FD_CLOEXEC** file descriptor flag associated with the new file descriptor is set.

The file status flags and file access modes of the open file description are according to the value of the *oflag* parameter. The *oflag* parameter is the bitwise-inclusive OR of the following flags defined in the **fcntl.h** header file.

Parameters

Item	Description
name	Points to a string naming a shared memory object.
oflag	Specifies the flags to be used by the **shm_open** subroutine.
mode	Sets the value of the permission bits of the shared memory object.

Read-Write Flags

Applications specify exactly one of the first two values (access modes) below in the value of the *oflag* parameter:

Item	Description
O_RDONLY	Open for read access only.
O_RDWR	Open for read or write access.

Other Flags

Any combination of the remaining flags may be specified in the value of the *oflag* parameter:

Item	Description
O_CREAT	If the shared memory object exists, this flag has no effect, except as noted under the **O_EXCL** flag below. Otherwise, the shared memory object is created, the user ID of the shared memory object is set to the effective user ID of the process, and the group ID of the shared memory object is set to the effective group ID of the process. The permission bits of the shared memory object are set to the value of the *mode* parameter except those set in the file mode creation mask of the process. Only the low-order 9 bits of the *mode* parameter are taken into account.The shared memory object has a size of zero.
O_EXCL	If the **O_EXCL** and **O_CREAT** flags are set, the **shm_open** subroutine fails if the shared memory object exists. The **O_EXCL** flag is ignored if the **O_CREAT** flag is not set.
O_TRUNC	If the shared memory object exists and it is successfully opened, the **O_RDWR** flag, the object is truncated to zero length, and the mode and owner is unchanged by the **shm_open** call.

Return Values

Upon successful completion, the **shm_open** subroutine returns a non-negative integer representing the lowest numbered unused file descriptor. If unsuccessful, it returns -1 and sets **errno** to indicate the error.

Error Codes

Item	Description
EACCES	The shared memory object exists and the permissions specified by the *oflag* parameter are denied, or the shared memory object does not exist and permission to create the shared memory object is denied, or the **O_TRUNC** flag is specified and write permission is denied.
EEXIST	The **O_CREAT** and **O_EXCL** flags are set and the named shared memory object already exists.
EINVAL	The **shm_open** subroutine is not supported for an empty name string, or the *name* parameter is missing, or the *oflag* parameter contains an invalid value.
EFAULT	The *name* parameter points outside of the allocated address space of the process.
EMFILE	Too many file descriptors are currently in use by this process.
ENAMETOOLONG	The length of the *name* parameter exceeds PATH_MAX or a pathname component is longer than NAME_MAX.
ENFILE	Too many shared memory objects are currently open in the system.
ENOENT	The **O_CREAT** flag is not set and the named shared memory object does not exist.
ENOMEM	The system is unable to allocate resources.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.
ENOSPC	There is insufficient space for the creation of the new shared memory object

shm_unlink Subroutine

Purpose

Removes a shared memory object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/mman.h>

int shm_unlink (name)
const char *name;
```

Description

The **shm_unlink** subroutine removes the name of the shared memory object named by the string pointed to by the *name* parameter.

If one or more references to the shared memory object exist when the object is unlinked, the name is removed before the **shm_unlink** subroutine returns, but the removal of the memory object contents is postponed until all open and map references to the shared memory object have been removed.

Even if the object continues to exist after the last **shm_unlink** call, reuse of the name subsequently causes the **shm_open** subroutine to behave as if no shared memory object of this name exists. In other words, the **shm_open** subroutine will fail if **O_CREAT** is not set, or will create a new shared memory object if **O_CREAT** is set.

Parameters

Item	Description
name	Specifies the name of the shared memory object to be unlinked.

Return Values

Upon successful completion, zero is returned. Otherwise, -1 is returned and **errno** is set to indicate the error. If -1 is returned, the named shared memory object is not changed by the subroutine call.

Error Codes

The **shm_unlink** subroutine fails if:

Item	Description
EACCES	Permission is denied to unlink the named shared memory object.
EFAULT	The *name* parameter points outside of the allocated address space of the process.
EINVAL	The name parameter is an empty name string, or is missing.
ENAMETOOLONG	The length of the *name* parameter exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.
ENOENT	The named shared memory object does not exist.
ENOTSUP	This function is not supported with processes that have been checkpoint-restart'ed.

shmat Subroutine

Purpose

Attaches a shared memory segment or a mapped file to the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/shm.h>
```

void *shmat (*SharedMemoryID***,** *SharedMemoryAddress***,** *SharedMemoryFlag***)**
int *SharedMemoryID*, *SharedMemoryFlag*;
const void * *SharedMemoryAddress*;

Description

The **shmat** subroutine attaches the shared memory segment or mapped file specified by the *SharedMemoryID* parameter (returned by the **shmget** subroutine), or file descriptor specified by the *SharedMemoryID* parameter (returned by the **openx** subroutine) to the address space of the calling process.

A call to the **shmat** subroutine on a file descriptor that identifies an open shared memory object fails with **EINVAL**.

To learn more about the limits that apply to shared memory, see the Inter-Process Communication (IPC) Limits article in *General Programming Concepts*.

An extended **shmat** capability is available. If an environment variable **EXTSHM=ON** is defined then processes executing in that environment will be able to create and attach more than eleven shared memory segments.

The segments can be of size from 1 byte to 2 GB. The process can attach segments larger than 256MB into the address space for the size of the segment. Another segment could be attached at the end of the first one in the same 256MB segment region. The address at which a process can attach is at page boundaries - a multiple of **SHMLBA_EXTSHM** bytes. For segments larger than 256MB in size, if **EXTSHM=ON** is not defined, the address at which a process can attach is at 256MB boundaries, which is a multiple of **SHMLBA** bytes.

The segments can be of size from 1 byte to 256MB. The process can attach these segments into the address space for the size of the segment. Another segment could be attached at the end of the first one in the same 256MB segment region. The address at which a process can attach will be at page boundaries - a multiple of **SHMLBA_EXTSHM** bytes.

The maximum address space available for shared memory with or without the environment variable and for memory mapping is 2.75GB. An additional segment register "0xE" is available so that the address space is from 0x30000000 to 0xE0000000. However, a 256MB region starting from 0xD0000000 will be used by the shared libraries and is therefore unavailable for shared memory regions or *mmap*ped regions.

On a 32-bit process running with the very large address space model has up to 3.25 GB of address space available for the **shmat** and **mmap** memory mappings. For a 32-bit process with the very large address space model, the address space available for mappings is from 0x30000000 to 0xFFFFFFFF. This extended address range applies to both extended **shmat** and standard **shmat**. For more information on how to use the very large address space model, see the Understanding the Very Large Address-Space Model article in *General Programming Concepts*.

There are some restrictions on the use of the extended shmat feature. These shared memory regions can not be used as I/O buffers where the unpinning of the buffer occurs in an interrupt handler. The restrictions on the use are the same as that of *mmap* buffers.

The smaller region sizes are not supported for mapping files. Regardless of whether **EXTSHM=ON** or not, mapping a file will consume at least 256MB of address space.

The **SHM_SIZE shmctl** command is not supported for segments created with **EXTSHM=ON**.

A segment created with **EXTSHM=ON** can be attached by a process without **EXTSHM=ON**. This will consume an area of address space that is a multiple of 256MB in size, regardless of the size of the shared memory region.

A segment created without **EXTSHM=ON** can be attached by a process with **EXTSHM=ON**. This will consume an area of address space that is a multiple of 256MB in size, regardless of the size of the shared memory region.

The environment variable provides the option of executing an application either with the additional functionality of attaching more than 11 segments when **EXTSHM=ON**, or the higher-performance access to 11 or fewer segments when the environment variable is not set.

The EXTSHM environment variable supports two additional values, EXTSHM=1SEG and EXTSHM=MSEG. All three options let users create more than 11 segments.

The EXTSHM=1SEG option defaults to the same behavior as EXTSHM=ON, which is to make memory mapped segments (type MMAP) of shared memories less than 256 MB, and SHMAT'ed segments (type WORKING) of shared memories greater than or equal to 256 MB. The EXTSHM=MSEG option creates memory mapped segments of all shared memories, regardless of size. This option provides better use of memory space.

Parameters

Item	Description
SharedMemoryID	Specifies an identifier for the shared memory segment.
SharedMemoryAddress	Identifies the segment or file attached at the address specified by the *SharedMemoryAddress* parameter, as follows: • If the *SharedMemoryAddress* parameter is not equal to 0, and the **SHM_RND** flag is set in the *SharedMemoryFlag* parameter, the segment or file is attached at the next lower segment boundary. This address is given by (*SharedMemoryAddress* -(*SharedMemoryAddress* modulo **SHMLBA_EXTSHM** if environment variable **EXTSHM=ON** or **SHMLBA** if not). SHMLBA specifies the low boundary address multiple of a segment. • If the *SharedMemoryAddress* parameter is not equal to 0 and the **SHM_RND** flag is not set in the *SharedMemoryFlag* parameter, the segment or file is attached at the address given by the *SharedMemoryAddress* parameter. If this address does not point to a **SHMLBA_EXTSHM** boundary if the environment variable **EXTSHM=ON** or **SHMLBA** boundary if not, the **shmat** subroutine returns the value -1 and sets the **errno** global variable to the **EINVAL** error code. SHMLBA specifies the low boundary address multiple of a segment.
SharedMemoryFlag	Specifies several options. Its value is either 0 or is constructed by logically ORing one or more of the following values: **SHM_COPY** Changes an open file to deferred update (see the **openx** subroutine). Included only for compatibility with previous versions of the operating system. **SHM_MAP** Maps a file onto the address space instead of a shared memory segment. The *SharedMemoryID* parameter must specify an open file descriptor in this case. **SHM_RDONLY** Specifies read-only mode instead of the default read-write mode. **SHM_RND** Rounds the address given by the *SharedMemoryAddress* parameter to the next lower segment boundary, if necessary.

The **shmat** subroutine makes a shared memory segment addressable by the current process. The segment is attached for reading if the **SHM_RDONLY** flag is set and the current process has read permission. If the **SHM_RDONLY** flag is not set and the current process has both read and write permission, it is attached for reading and writing.

If the **SHM_MAP** flag is set, file mapping takes place. In this case, the **shmat** subroutine maps the file open on the file descriptor specified by the *SharedMemoryID* onto a segment. The file must be a regular file. The segment is then mapped into the address space of the process. A file of any size can be mapped if there is enough space in the user address space.

When file mapping is requested, the *SharedMemoryFlag* parameter specifies how the file should be mapped. If the **SHM_RDONLY** flag is set, the file is mapped read-only. To map read-write, the file must have been opened for writing.

All processes that map the same file read-only or read-write map to the same segment. This segment remains mapped until the last process mapping the file closes it.

A mapped file opened with the **O_DEFER** update has deferred update. That is, changes to the shared segment do not affect the contents of the file resident in the file system until an **fsync** subroutine is issued to the file descriptor for which the mapping was requested. Setting the **SHM_COPY** flag changes the file to the deferred state. The file remains in this state until all processes close it. The **SHM_COPY** flag is provided only for compatibility with Version 2 of the operating system. New programs should use the **O_DEFER** open flag.

A file descriptor can be used to map the corresponding file only once. To map a file several times requires multiple file descriptors.

When a file is mapped onto a segment, the file is referenced by accessing the segment. The memory paging system automatically takes care of the physical I/O. References beyond the end of the file cause the file to be extended in page-sized increments. The file cannot be extended beyond the next segment boundary.

Attention: When a file is mapped, use of standard file system calls, such as **truncate** and **write**, are discouraged and might produce unexpected results, especially in a multithreaded environment. In particular, the **write** system call, upon completion, sets the size to the new end-of-file. Any **shmat** changes that occur concurrently past this new end-of-file might be lost. Concurrent change of the mapped region and use of the **write** system call are highly discouraged.

Return Values

When successful, the segment start address of the attached shared memory segment or mapped file is returned. Otherwise, the shared memory segment is not attached, the **errno** global variable is set to indicate the error, and a value of -1 is returned.

Error Codes

The **shmat** subroutine is unsuccessful and the shared memory segment or mapped file is not attached if one or more of the following are true:

Item	Description
EACCES	The calling process is denied permission for the specified operation.
EAGAIN	The file to be mapped has enforced locking enabled, and the file is currently locked.
EBADF	A file descriptor to map does not refer to an open regular file.
EEXIST	The file to be mapped has already been mapped.
EINVAL	The **SHM_RDONLY** and **SHM_COPY** flags are both set.
EINVAL	The **shmat** subroutine was used with a file descriptor obtained from a call to the **shm_open** subroutine.
EINVAL	The *SharedMemoryID* parameter is not a valid shared memory identifier.

Item	Description
EINVAL	The *SharedMemoryAddress* parameter is not equal to 0, and the value of (*SharedMemoryAddress* - (*SharedMemoryAddress* modulo **SHMLBA_EXTSHM** if the environment variable **EXTSHM=ON** or **SHMLBA** if not) points outside the address space of the process.
EINVAL	The *SharedMemoryAddress* parameter is not equal to 0, the **SHM_RND** flag is not set in the *SharedMemoryFlag* parameter, and the *SharedMemoryAddress* parameter points to a location outside of the address space of the process.
EMFILE	The number of shared memory segments attached to the calling process exceeds the system-imposed limit.
ENOMEM	The available data space in memory is not large enough to hold the shared memory segment. ENOMEM is always returned if a 32-bit process tries to attach a shared memory segment larger than 2GB.
ENOMEM	The available data space in memory is not large enough to hold the mapped file data structure.

shmctl Subroutine

Purpose

Controls shared memory operations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/shm.h>

int shmctl (SharedMemoryID, Command, Buffer)
int  SharedMemoryID, Command;
struct shmid_ds * Buffer;
```

Description

The **shmctl** subroutine performs a variety of shared-memory control operations as specified by the *Command* parameter.

The following limits apply to shared memory:

- Minimum shared-memory segment size is 64 GB for 64-bit applications.
- Maximum number of shared memory IDs is 131072.

Parameters

Item	Description
SharedMemoryID	Specifies an identifier returned by the **shmget** subroutine.
Buffer	Indicates a pointer to the **shmid_ds** structure. The **shmid_ds** structure is defined in the **sys/shm.h** file.

Item	Description

Command	The following commands are available:
	IPC_STAT
	Obtains status information about the shared memory segment identified by the *SharedMemoryID* parameter. This information is stored in the area pointed to by the *Buffer* parameter. The calling process must have read permission to run this command. The `shm_pagesize` and `shm_lba` fields of the **shmid_ds** data structure pointed to by the Buffer parameter are not updated by this command.
	IPC_ SET
	Sets the user and group IDs of the owner as well as the access permissions for the shared memory segment identified by the *SharedMemoryID* parameter. This command sets the following fields:
	`shm_perm.uid  /* owning user ID       */`
	`shm_perm.gid  /* owning group ID      */`
	`shm_perm.mode /* permission bits only */`
	You must have an effective user ID equal to root or to the value of the `shm_perm.cuid` or `shm_perm.uid` field in the **shmid_ds** data structure identified by the *SharedMemoryID* parameter.
	IPC_RMID
	Removes the shared memory identifier specified by the *SharedMemoryID* parameter from the system and erases the shared memory segment and data structure associated with it. This command is only executed by a process that has an effective user ID equal either to that of `superuser` or to the value of the `shm_perm.uid` or `shm_perm.cuid` field in the data structure identified by the *SharedMemoryID* parameter.
	SHM_SIZE
	Sets the size of the shared memory segment to the value specified by the `shm_segsz` field of the structure specified by the *Buffer* parameter. This value can be larger or smaller than the current size. The limit is the maximum shared-memory segment size. This command is only executed by a process that has an effective user ID equal either to that of a process with the appropriate privileges or to the value of the `shm_perm.uid` or `shm_perm.cuid` field in the data structure identified by the *SharedMemoryID* parameter. This command is not supported for regions created with the environment variable **EXTSHM=ON**. This results in a return value of -1 with **errno** set to **EINVAL**. Attempting to use the **SHM_SIZE** on a shared memory region larger than 256MB or attempting to increase the size of a shared memory region larger than 256MB results in a return value of -1 with **errno** set to **EINVAL**.

Item	Description
	SHM_BSR Backs the shared memory region identified by the *SharedMemoryID* parameter with barrier synchronization register (BSR) memory. BSR shared memory can be used for efficiently implementing barrier synchronization constructs that are commonly used in highly parallel workloads. The Buffer parameter must be set to **NULL** when using this command. This command can only be used by a process that has an effective user ID equal to that of superuser or to the value of the shm_perm.uid or shm_perm.cuid fields in the **shmid_ds** data structure identified by the *SharedMemoryID*parameter. A non-root user must have the **CAP_BYPASS_RAC_VMM** capability in order to allocate BSR memory and **PV_KER_RAC** privilege if using RBAC. If insufficient BSR memory is available to satisfy the request, **shmctl()** will fail with **errno** set to **ENOMEM**. In order to use BSR memory for a shared memory region, this command must be used on the shared memory region immediately after it has been created and before any process has attached to the shared memory region. This command cannot be used with shared memory regions that have been created with the **SHM_PIN** flag or shared memory regions that have been locked with the **SHM_LOCK shmctl()** command. This command also cannot be used on shared memory regions whose page size has been changed with the SHM_PAGESIZE shmctl() command, as well as shared memory regions created with the **EXTSHM=ON** environment variable. **SHM_PAGESIZE** Sets the page size backing the shared memory segment identified by the *SharedMemoryID* parameter. This command will set the page size backing the specified shared memory segment to the value of the `shm_pagesize` field of the **shmid_ds** structure specified by the *Buffer* parameter. The `shm_pagesize` field is interpreted as a page size in bytes. This command can only be used by a process that has an effective user ID with permissions set equal either to that of superuser or to the value of the `shm_perm.uid` or `shm_perm.cuid` field in the **shmid_ds** data structure identified by the *SharedMemoryID* parameter. In order to change the page size backing a shared memory segment, this command must be used on the shared memory segment immediately after it has been created and before any process has attached to the shared memory segment. Also, this command must be used before pinning the pages in a shared memory segment. Thus, this command cannot be used with shared memory segments that have been created with the **SHM_PIN** flag or shared memory segments that have been pinned with the **SHM_LOCK shmctl()** command. This command cannot be used with shared memory regions created with the **EXTSHM=ON** environment variable. **Note:** A system's supported page sizes can be queried by specifying the **VM_GETPSIZES** command to the **vmgetinfo()** system call.
Command continued	The following commands are available: **SHM_LOCK** Pins all of the pages in the shared memory segment identified by the *SharedMemoryID* parameter. Pinning the pages in a shared memory segment will ensure that page faults do not occur for memory references to the shared memory region. This command can only be used by a process that has an effective user ID equal to that of superuser or to the value of the `shm_perm.uid` or `shm_perm.cuid` field in the **shmid_ds** data structure identified by the *SharedMemoryID* parameter. A non-superuser user must also have the **CAP_BYPASS_RAC_VMM** capability in order to use this command. This command cannot be used with shared memory regions created with the **EXTSHM=ON** environment variable or shared memory regions created with the **SHM_PIN** flag. The Buffer parameter must be set to **NULL** when using this command.

Item	Description
	SHM_UNLOCK Unpins all of the pages in the shared memory segment identified by the *SharedMemoryID* parameter. This command can only be used by a process that has an effective user ID equal either to that of superuser or to the value of the `shm_perm.uid` or `shm_perm.cuid` field in the **shmid_ds** data structure identified by the `SharedMemoryID` parameter. This command will fail if called on shared memory segments created with the **SHM_PIN** flag. Also, this command can only be used when the specified shared memory segment is not attached by any process, and there is no outstanding I/O to the shared memory segment. The Buffer parameter must be set to **NULL** when using this command. **SHM_GETLBA** Obtains the minimum alignment of the address at which the shared memory segment identified by the *SharedMemoryID* parameter can be attached by the **shmat()** subroutine. This command will store the minimum alignment in the `shm_lba` field of the **shmid_ds** struct pointed to by the Buffer parameter. The alignment is reported in bytes. The calling process must have read permission to a shared memory region in order to use this command.

Return Values

When completed successfully, the **shmctl** subroutine returns a value of 0. Otherwise, it returns a value of -1 and the **errno** global variable is set to indicate the error.

Error Codes

The **shmctl** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EACCES	The *Command* parameter is equal to the **IPC_STAT** or **SHM_GETLBA** value and read permission is denied to the calling process.
EFAULT	The *Buffer* parameter points to a location outside the allocated address space of the process.
EINVAL	The *SharedMemoryID* parameter is not a valid shared memory identifier.
EINVAL	The *Command* parameter is not a valid command.
EINVAL	The *Command* parameter is equal to the **SHM_SIZE** value and the value of the `shm_segsz` field of the structure specified by the *Buffer* parameter is not valid.
EINVAL	The *Command* parameter is equal to the **SHM_SIZE**, **SHM_PAGESIZE**, **SHM_LOCK** or **SHM_BSR** value and the shared memory region was created with the environment variable **EXTSHM=ON**.
EINVAL	The Command parameter is equal to the **SHM_PAGESIZE** value and the value of the `shm_pagesize` field of the structure specified by the *Buffer* parameter is not supported.
EINVAL	The Command parameter is equal to **SHM_UNLOCK**, and the specified shared memory segment was not previously locked by a **SHM_LOCK** operation.
EINVAL	The Command parameter is equal to **SHM_LOCK SHM_UNLOCK**, or **SHM_BSR** and the Buffer parameter is not **NULL**.

Item	Description
EINVAL	The Command parameter is equal to **SHM_BSR**, and the shared memory region's page size has previously been changed via the **SHM_PAGESIZE** command.
ENOMEM	The Command parameter is **SHM_BSR**, and there is insufficient **BSR** memory available to back the entire shared memory segment.
ENOMEM	The *Command* parameter is equal to the **SHM_SIZE** value, and the attempt to change the segment size is unsuccessful because the system does not have enough memory.
ENOMEM	The Command parameter is **SHM_LOCK**, and locking the pages in the specified shared memory segment would exceed the limit on the amount of memory the calling process may lock.
ENOMEM	The Command parameter is **SHM_PAGESIZE**, and there are insufficient pages of the specified page size to back the entire shared memory segment.
EOVERFLOW	The *Command* parameter is **IPC_STAT** and the size of the shared memory region is greater than or equal to 4G bytes. This only happens with 32-bit programs.
EPERM	The *Command* parameter is **IPC_RMID**, **SHM_SIZE**, **SHM_PAGESIZE**, **SHM_LOCK**, or **SHM_UNLOCK**, and the effective user ID of the calling process is not equal to the value of the `shm_perm.uid` or `shm_perm.cuid` field in the data structure identified by the *SharedMemoryID* parameter. The effective user ID of the calling process is not the root user ID.
EPERM	The *Command* parameter is **SHM_PAGESIZE**, and the calling process does not have the appropriate privilege to allocate pages of the specified page size.
EPERM	The *Command* parameter is **SHM_LOCK SHM_UNLOCK**, or **SHM_BSR** and the calling process does not have the appropriate privilege to perform the requested operation.
EBUSY	The *Command* parameter is **SHM_LOCK** or **SHM_UNLOCK**, and the specified shared memory segment is currently being used for I/O or is attached by one or more processes.
EBUSY	The *Command* parameter is **SHM_PAGESIZE** or **SHM_BSR** and the specified shared memory segment has already been attached by one or more processes or has been pinned via **SHM_PIN** or **SHM_LOCK**.

Examples

The following example allocates a 32MB shared memory region, changes the page size for the shared memory region to 64K, and then pins all of the pages in the shared memory region:

```
int    id;
size_t shm_size;
struct shmid_ds shm_buf = { 0 };
psize_t psize_64k;

psize_64k = 64 * 1024;

/* Create a 32MB shared memory region */
shm_size = 32*1024*1024;

/* Allocate the shared memory region */
if ((id = shmget(IPC_PRIVATE, shm_size, IPC_CREAT)) < 0)
{
    perror("shmget() failed");
```

```
    return -1;
}

/* Use 64K pages for the shared memory region */
shm_buf.shm_pagesize = psize_64k;
if (shmctl(id, SHM_PAGESIZE, &shm_buf))
{
    perror("shmctl(SHM_PAGESIZE) failed");
}

/* Pin all of the pages in the shared memory region */
if (shmctl(id, SHM_LOCK, NULL))
{
    perror("shmctl(SHM_LOCK) failed");
}
```

The following example allocates a 16MB shared memory region and determines the minimum alignment of the address at which an application can **shmat()** the shared memory region:

```
int    id;
size_t shm_size;
struct shmid_ds shm_buf = { 0 };

/* Create a 16MB shared memory region */
shm_size = 16*1024*1024;

/* Allocate the shared memory region */
if ((id = shmget(IPC_PRIVATE, shm_size, IPC_CREAT)) < 0)
{
    perror("shmget() failed");
    return -1;
}

/* Determine the address alignment requirements */
if (shmctl(id, SHM_GETLBA, &shm_buf))
{
    perror("shmctl(SHM_GETLBA) failed");
}
else
{
    printf("shmlba = %08llx\n", shm_buf.shm_lba);
}
```

shmdt Subroutine

Purpose

Detaches a shared memory segment.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/shm.h>
```

int shmdt (*SharedMemoryAddress*)
const void * *SharedMemoryAddress*;

Description

The **shmdt** subroutine detaches from the data segment of the calling process the shared memory segment located at the address specified by the *SharedMemoryAddress* parameter.

Mapped file segments are automatically detached when the mapped file is closed. However, you can use the **shmdt** subroutine to explicitly release the segment register used to map a file. Shared memory segments must be explicitly detached with the **shmdt** subroutine.

If the file was mapped for writing, the **shmdt** subroutine updates the **mtime** and **ctime** time stamps.

The following limits apply to shared memory:

- Maximum shared-memory segment size is 64 GB for 64-bit applications.
- Minimum shared-memory segment size is 1 byte.
- Maximum number of shared memory IDs is 131072.

Parameters

Item	Description
SharedMemoryAddress	Specifies the data segment start address of a shared memory segment.

Return Values

When successful, the **shmdt** subroutine returns a value of 0. Otherwise, the shared memory segment at the address specified by the *SharedMemoryAddress* parameter is not detached, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **shmdt** subroutine is unsuccessful if the following condition is true:

Item	Description
EINVAL	The value of the *SharedMemoryAddress* parameter is not the data-segment start address of a shared memory segment.

shmget Subroutine

Purpose

Gets shared memory segments.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/shm.h>

int shmget (Key, Size, SharedMemoryFlag)
key_t  Key;
size_t  Size
int  SharedMemoryFlag;
```

Description

The **shmget** subroutine returns the shared memory identifier associated with the specified *Key* parameter.

The following limits apply to shared memory:

- Maximum shared-memory segment size is 64 GB for 64-bit applicatoins.
- Minimum shared-memory segment size is 1 byte.
- Maximum number of shared memory IDs is 131072.

Parameters

Item	Description
Key	Specifies either the **IPC_PRIVATE** value or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm).
Size	Specifies the number of bytes of shared memory required.
SharedMemoryFlag	Constructed by logically ORing one or more of the following values: **IPC_CREAT** Creates the data structure if it does not already exist. **IPC_EXCL** Causes the **shmget** subroutine to be unsuccessful if the **IPC_CREAT** flag is also set, and the data structure already exists. **SHM_LGPAGE** Attempts to create the region so it can be mapped through hardware-supported, large-page mechanisms, if enabled. This is purely advisory. For the system to consider this flag, it must be used in conjunction with the **SHM_PIN** flag and enabled with the **vmtune** command (-L to reserve memory for the region (which requires a reboot) and -S to enable **SHM_PIN**). To successfully get large-pages, the user requesting large-page shared memory must have **CAP_BYPASS_RAC_VMM** capability. This has no effect on shared memory regions created with the EXTSHM=ON environment variable. **SHM_PIN** Attempts to pin the shared memory region if enabled. This is purely advisory. For the system to consider this flag, the system must be enable with **vmtune** command. This has no effect on shared memory regions created with EXTSHM=ON environment variable. **S_IRUSR** Permits the process that owns the data structure to read it. **S_IWUSR** Permits the process that owns the data structure to modify it. **S_IRGRP** Permits the group associated with the data structure to read it. **S_IWGRP** Permits the group associated with the data structure to modify it. **S_IROTH** Permits others to read the data structure. **S_IWOTH** Permits others to modify the data structure. Values that begin with the **S_I** prefix are defined in the **sys/mode.h** file and are a subset of the access permissions that apply to files.

A shared memory identifier, its associated data structure, and a shared memory segment equal in number of bytes to the value of the *Size* parameter are created for the *Key* parameter if one of the following is true:

- The *Key* parameter is equal to the **IPC_PRIVATE** value.
- The *Key* parameter does not already have a shared memory identifier associated with it, and the **IPC_CREAT** flag is set in the *SharedMemoryFlag* parameter.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- The `shm_perm.cuid` and `shm_perm.uid` fields are set to the effective user ID of the calling process.
- The `shm_perm.cgid` and `shm_perm.gid` fields are set to the effective group ID of the calling process.

- The low-order 9 bits of the `shm_perm.mode` field are set to the low-order 9 bits of the *SharedMemoryFlag* parameter.
- The `shm_segsz` field is set to the value of the *Size* parameter.
- The `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` fields are set to 0.
- The `shm_ctime` field is set to the current time.

Note: Once created, a shared memory segment is deleted only when the system reboots or by issuing the **ipcrm** command or using the following **shmctl** subroutine:

```
if (shmctl (id, IPC_RMID, 0) == -1)
 perror ("error in closing segment"),exit (1);
```

Return Values

Upon successful completion, a shared memory identifier is returned. Otherwise, the **shmget** subroutine returns a value of -1 and sets the **errno** global variable to indicate the error.

Error Codes

The **shmget** subroutine is unsuccessful if one or more of the following are true:

Item	Description
EACCES	A shared memory identifier exists for the *Key* parameter, but operation permission as specified by the low-order 9 bits of the *SharedMemoryFlag* parameter is not granted.
EEXIST	A shared memory identifier exists for the *Key* parameter, and both the **IPC_CREAT** and **IPC_EXCL** flags are set in the *SharedMemoryFlag* parameter.
EINVAL	A shared memory identifier does not exist and the *Size* parameter is less than the system-imposed minimum or greater than the system-imposed maximum.
EINVAL	A shared memory identifier exists for the *Key* parameter, but the size of the segment associated with it is less than the *Size* parameter, and the *Size* parameter is not equal to 0.
ENOENT	A shared memory identifier does not exist for the *Key* parameter, and the **IPC_CREAT** flag is not set in the *SharedMemoryFlag* parameter.
ENOMEM	A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to meet the request.
ENOSPC	A shared memory identifier will be created, but the system-imposed maximum of shared memory identifiers allowed will be exceeded.

sigaction, sigvec, or signal Subroutine

Purpose

Specifies the action to take upon delivery of a signal.

Libraries

Item	Description
sigaction	Standard C Library (**libc.a**)
signal, sigvec	Standard C Library (**libc.a**);

Berkeley Compatibility Library (**libbsd.a**)

Syntax

```
#include <signal.h>

int sigaction ( signal,  action,  oaction)
int signal;
struct sigaction *action, *oaction;

int sigvec (signal,  invec,  outvec)
int signal;
struct sigvec *invec, *outvec;

void (*signal (signal, action)) ()
int signal;
void (*action) (int);
```

Description

The **sigaction** subroutine allows a calling process to examine and change the action to be taken when a specific signal is delivered to the process issuing this subroutine.

In multi-threaded applications using the threads library (**libpthreads.a**), signal actions are common to all threads within the process. Any thread calling the **sigaction** subroutine changes the action to be taken when a specific signal is delivered to the threads process, that is, to any thread within the process.

Note: The **sigaction** subroutine must not be used concurrently to the **sigwait** subroutine on the same signal.

The *signal* parameter specifies the signal. If the *action* parameter is not null, it points to a **sigaction** structure that describes the action to be taken on receipt of the *signal* parameter signal. If the *oaction* parameter is not null, it points to a **sigaction** structure in which the signal action data in effect at the time of the **sigaction** subroutine call is returned. If the *action* parameter is null, signal handling is unchanged; thus, the call can be used to inquire about the current handling of a given signal.

The **sigaction** structure has the following fields:

Member Type	Member Name	Description
void(*) (int)	sa_handler	SIG_DFL, SIG_IGN or pointer to a function.
sigset_t	sa_mask	Additional set of signals to be blocked during execution of signal-catching function.
int	sa_flags	Special flags to affect behaviour of signal.
void(*) (int, siginfo_t *, void *)	sa_sigaction	Signal-catching function.

The `sa_handler` field can have a **SIG_DFL** or **SIG_IGN** value, or it can be a pointer to a function. A **SIG_DFL** value requests default action to be taken when a signal is delivered. A value of **SIG_IGN** requests that the signal have no effect on the receiving process. A pointer to a function requests that the signal be caught; that is, the signal should cause the function to be called. These actions are more fully described in "Parameters".

When a signal is delivered to a thread, if the action of that signal specifies termination, stop, or continue, the entire process is terminated, stopped, or continued, respectively.

If the SA_SIGINFO flag (see below) is cleared in the `sa_flags` field of the **sigaction** structure, the `sa_handler` field identifies the action to be associated with the specified signal. If the SA_SIGINFO flag is set in the `sa_flags` field, the `sa_sigaction` field specifies a signal-catching function. If the

SA_SIGINFO bit is cleared and the `sa_handler` field specifies a signal-catching function, or if the SA_SIGINFO bit is set, the `sa_mask` field identifies a set of signals that will be added to the signal mask of the thread before the signal-catching function is invoked.

The `sa_mask` field can be used to specify that individual signals, in addition to those in the process signal mask, be blocked from being delivered while the signal handler function specified in the `sa_handler` field is operating. The `sa_flags` field can have the **SA_ONSTACK**, **SA_OLDSTYLE**, or **SA_NOCLDSTOP** bits set to specify further control over the actions taken on delivery of a signal.

If the **SA_ONSTACK** bit is set, the system runs the signal-catching function on the signal stack specified by the **sigstack** subroutine. If this bit is not set, the function runs on the stack of the process to which the signal is delivered.

If the **SA_OLDSTYLE** bit is set, the signal action is set to **SIG_DFL** label prior to calling the signal-catching function. This is supported for compatibility with old applications, and is not recommended since the same signal can recur before the signal-catching subroutine is able to reset the signal action and the default action (normally termination) is taken in that case.

If a signal for which a signal-catching function exists is sent to a process while that process is executing certain subroutines, the call can be restarted if the **SA_RESTART** bit is set for each signal. The only affected subroutines are the following:

- **read**,**readx**, **readv**, or **readvx** ("read, readx, read64x, readv, readvx, eread, ereadv, pread, or preadv Subroutine" on page 1714)
- **write**,**writex**, **writev**, or **writevx** ("write, writex, write64x, writev, writevx, ewrite, ewritev, pwrite, or pwritev Subroutine" on page 2365)
- **ioctl** or **ioctlx**
- **fcntl**, **lockf**, or **flock**
- **wait**, **wait3**, or **waitpid** ("wait, waitpid, wait3, wait364, and wait4 Subroutine" on page 2293)

Other subroutines do not restart and return **EINTR** label, independent of the setting of the **SA_RESTART** bit.

If **SA_SIGINFO** is cleared and the signal is caught, the signal-catching function will be entered as: `void func(int signo);`

Where *signo* is the only argument to the signal catching function. In this case the **sa_handler** member must be used to describe the signal catching function and the application must not modify the **sa_sigaction** member. If SA_SIGINFO is set and the signal is caught, the signal-catching function will be entered as: `void func(int signo, siginfo_t * info, void * context);` where two additional arguments are passed to the signal catching function.

The second argument will point to an object of type **siginfo_t** explaining the reason why the signal was generated. The third argument can be cast to a pointer to an object of type **ucontext_t** to refer to the receiving process' context that was interrupted when the signal was delivered. In this case the **sa_sigaction** member must be used to describe the signal catching function and the application must not modify the **sa_handler** member.

The **si_signo** member contains the system-generated signal number. The **si_errno** member may contain implementation-dependent additional error information. If nonzero, it contains an error number identifying the condition that caused the signal to be generated. The **si_code** member contains a code identifying the cause of the signal. If the value of **si_code** is less than or equal to **0**, the signal was generated by a process and **si_pid** and **si_uid** respectively indicate the process ID and the real user ID of the sender.

The **signal.h** header description contains information about the signal specific contents of the elements of the **siginfo_t** type. If **SA_NOCLDWAIT** is set and **sig** equals SIGCHLD, child processes of the calling processes will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and **wait**, **wait3**, **waitid** and **waitpid** will fail and set **errno** to ECHILD. Otherwise, terminating child processes will be transformed into zombie

processes, unless **SIGCHLD** is set to **SIG_IGN**. When SIGCHLD is set to SIG_IGN, the signal is ignored and any zombie children of the process will be cleaned up.

If **SA_RESETHAND** is set, the disposition of the signal will be reset to **SIG_DFL** and the **SA_SIGINFO** flag will be cleared on entry to the signal handler.

If **SA_NODEFER** is set and *sig* is caught, *sig* will not be added to the process' signal mask on entry to the signal handler unless it is included in **sa_mask**. Otherwise, *sig* will always be added to the process' signal mask on entry to the signal handler. If *sig* is SIGCHLD, the SA_NOCLDSTOP flag is not set in sa_flags, and the implementation supports the SIGCHLD signal, a SIGCHLD signal will be generated for the calling process whenever any of its child processes stop.

If *sig* is SIGCHLD and the SA_NOCLDSTOP flag is set in sa_flags, the implementation will not generate a SIGCHLD signal in this way. When a signal is caught by a signal-catching function installed by **sigaction**, a new signal mask is calculated and installed for the duration of the signal-catching function (or until a call to either **sigprocmask orsigsuspend** is made).

This mask is formed by taking the union of the current signal mask and the value of the **sa_mask** for the signal being delivered unless SA_NODEFER or SA_RESETHAND is set, and including the signal being delivered. If the user's signal handler returns normally, the original signal mask is restored.

Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to **sigaction**), until the SA_RESETHAND flag causes resetting of the handler, or until one of the **exec** functions is called.

If the previous action for *sig* had been established by **signal**, the values of the fields returned in the structure pointed to by *oact* are unspecified, and in particular **oact->sa_handler** is not necessarily the same value passed to **signal**.

However, if a pointer to the same structure or a copy thereof is passed to a subsequent call to **sigaction** through the *act* argument, handling of the signal will be as if the original call to **signal** were repeated.

If **sigaction** fails, no new signal handler is installed. It is unspecified whether an attempt to set the action for a signal that cannot be caught or ignored to **SIG_DFL** is ignored or causes an error to be returned with **errno** set to **EINVAL**.

If **SA_SIGINFO** is not set in **sa_flags**, then the disposition of subsequent occurrences of sig when it is already pending is implementation-dependent; the signal-catching function will be invoked with a single argument.

The **sigvec** and **signal** subroutines are provided for compatibility to older operating systems. Their function is a subset of that available with **sigaction**.

The **sigvec** subroutine uses the **sigvec** structure instead of the **sigaction** structure. The **sigvec** structure specifies a mask as an `int` instead of a `sigset_t`. The mask for the **sigvec** subroutine is constructed by setting the `i-th` bit in the mask if signal *i* is to be blocked. Therefore, the **sigvec** subroutine only allows signals between the values of 1 and 31 to be blocked when a signal-handling function is called. The other signals are not blocked by the signal-handler mask.

The **sigvec** structure has the following members:

```
int  (*sv_handler)();
/* signal handler */
int  sv_mask;
/* signal mask    */
int  sv_flags;
/* flags          */
```

The **sigvec** subroutine in the **libbsd.a** library interprets the **SV_INTERRUPT** flag and inverts it to the **SA_RESTART** flag of the **sigaction** subroutine. The **sigvec** subroutine in the **libc.a** library always sets the **SV_INTERRUPT** flag regardless of what was passed in the **sigvec** structure.

The **signal** subroutine in the **libc.a** library allows an action to be associated with a signal. The *action* parameter can have the same values that are described for the `sv_handler` field in the **sigaction** structure of the**sigaction** subroutine. However, no signal handler mask or flags can be specified; the

signal subroutine implicitly sets the signal handler mask to additional signals and the flags to be **SA_OLDSTYLE**.

Upon successful completion of a **signal** call, the value of the previous signal action is returned. If the call fails, a value of -1 is returned and the **errno** global variable is set to indicate the error as in the **sigaction** call.

The **signal** in **libc.a** does not set the **SA_RESTART** flag. It sets the signal mask to the signal whose action is being specified, and sets flags to **SA_OLDSTYLE**. The Berkeley Software Distribution (BSD) version of **signal** sets the **SA_RESTART** flag and preserves the current settings of the signal mask and flags. The BSD version can be used by compiling with the Berkeley Compatibility Library (**libbsd.a**).

Parameters

signal
Defines the signal. The following list describes signal names and the specification for each. The value of the *signal* parameter can be any signal name from this list or its corresponding number except the **SIGKILL** name. If you use the signal name, you must include the **signal.h** file, because the name is correlated in the file with its corresponding number.

Note: The symbols in the following list of signals represent these actions:

*****
Specifies the default action that includes creating a core dump file.

@
Specifies the default action that stops the process receiving these signals.

!
Specifies the default action that restarts or continues the process receiving these signals.

+
Specifies the default action that ignores these signals.

%
Indicates a likely shortage of paging space.

#
See *Terminal Programming* for more information on the use of these signals.

reserved
(26)

reserved
(37-58)

SIGALRM
Alarm clock. (14)

SIGBUS
Specification exception. (10*)

SIGCHLD
To parent on child stop or exit. (20+)

SIGCONT
Continue if stopped. (19!)

SIGDANGER
Paging space low. (33+%)

SIGEMT
EMT instruction. (7*)

SIGFPE
Arithmetic exception, integer divide by 0, or floating-point exception. (8*)

SIGHUP
Hang-up. (1)

SIGILL
Invalid instruction (not reset when caught). (4*)

SIGINT
Interrupt. (2)

SIGIO
Input/output possible or completed. (23+)

SIGGRANT
Monitor access wanted. (60#)

SIGMIGRATE
Migrate process. (35)

SIGMSG
Input data has been stored into the input ring buffer. (27#)

SIGPRE
Programming exception (user defined). (36)

SIGPROF
Profiling timer expired. (see the **setitimer** subroutine).(32)

SIGPWR
Power-fail restart. (29+)

SIGQUIT
Quit. (3*)

SIGIOT
End process (see the **abort** subroutine). (6*)

SIGKILL
Kill (cannot be caught or ignored). (9)

SIGPIPE
Write on a pipe when there is no process to read it. (13)

SIGRETRACT
Monitor access should be relinquished. (61#)

SIGSAK
Secure attention key. (63)

SIGSEGV
Segmentation violation. (11*)

SIGSOUND
A sound control has completed execution. (62#)

SIGSTOP
Stop (cannot be caught or ignored). (17@)

SIGSYS
Parameter not valid to subroutine. (12*)

SIGTALRM
Thread alarm clock. (38)

SIGTERM
Software termination signal. (15)

SIGTRAP
Trace trap (not reset when caught). (5*)

SIGTSTP
Interactive stop. (18@)

SIGTTIN
Background read attempted from control terminal. (21@)

SIGTTOU
Background write attempted from control terminal. (22@)

SIGURG
Urgent condition on I/O channel. (16+)

SIGUSR1
User-defined signal 1. (30)

SIGUSR2
User-defined signal 2. (31)

SIGVTALRM
Virtual time alarm (see the **setitimer** subroutine). (34)

SIGWINCH
Window size change. (28+)

SIGXCPU
CPU time limit exceeded (see the **setrlimit** subroutine). (24)

SIGXFSZ
File size limit exceeded (see the **setrlimit** subroutine).(25)

action
Points to a **sigaction** structure that describes the action to be taken upon receipt of the *signal* parameter signal.

The three types of actions that can be associated with a signal (**SIG_DFL**, **SIG_IGN**, or a pointer to a function) are described as follows:

- **SIG_DFL** Default action: signal-specific default action.

  Except for those signal numbers marked with a **+** (plus sign), **@** (at sign), or **!** (exclamation point), the default action for a signal ends the receiving process with all of the consequences described in the **_exit** subroutine. In addition, a memory image file is created in the current directory of the receiving process if an asterisk appears with a *signal* parameter and the following conditions are met:

  – All dumped cores are in the context of the running process. They are dumped with an owner and a group matching the effective user ID (UID) and group ID (GID) of the process. If this UID/GID pair does not have permission to write to the target directory that is determined according to the standard core path procedures, no core file is dumped.
  – If the real user ID (RUID) is root, the core file is dumped, with a mode of 0600.
  – If the effective user ID (EUID) matches the real user ID (RUID), and the effective group ID (EGID) matches any group in the credential's group list, the core file is dumped with permissions of 0600.
  – If the EUID matches the RUID, but the EGID does not match any group in the credential's group list, the core file cannot be dumped. The effective user cannot see data that they do not have access to.
  – If the EUID does not match the RUID, the core file can be dumped only if you have set a core directory using the **syscorepath** command. This avoids dumping the core file into either the current working directory or a user-specific core directory in such a way that you cannot remove the core file. Core is dumped with a mode of 0600. If you have not used the **syscorepath** command to set a core directory, no core is dumped.

  For signal numbers marked with a **!** (exclamation point), the default action restarts the receiving process if it has stopped, or continues to run the receiving process.

  For signal numbers marked with a **@** (at sign), the default action stops the execution of the receiving process temporarily. When a process stops, a **SIGCHLD** signal is sent to its parent process, unless the parent process has set the **SA_NOCLDSTOP** bit. While a process has stopped, any additional signals that are sent are not delivered until the process has started again. An exception to this is the **SIGKILL** signal, which always terminates the receiving process. Another exception is the **SIGCONT**

signal, which always causes the receiving process to restart or continue running. A process whose parent process has ended is sent a **SIGKILL** signal if the **SIGTSTP**, **SIGTTIN**, or **SIGTTOU** signals are generated for that process.

For signal numbers marked with a **+**, the default action ignores the signal. In this case, the delivery of a signal does not affect the receiving process.

If a signal action is set to **SIG_DFL** while the signal is pending, the signal remains pending.

- **SIG_IGN** Ignore signal.

  Delivery of the signal does not affect the receiving process. If a signal action is set to the **SIG_IGN** action while the signal is pending, the pending signal is discarded.

  An exception to this is the **SIGCHLD** signal whose **SIG_DFL** action ignores the signal. If the action for the **SIGCHLD** signal is set to **SIG_IGN**, child processes of the calling processes will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate, and **wait**, **wait3**, **waitid** and **waitpid** will fail and set **errno** to **ECHILD**.

  Note: The **SIGKILL** and **SIGSTOP** signals cannot be ignored.

- Pointer to a function, catch signal.

  Upon delivery of the signal, the receiving process runs the signal-catching function specified by the pointer to function. The signal-handler subroutine can be declared as follows:

```
handler(signal, Code, SCP)
int signal, Code;
struct sigcontext *SCP;
```

  The *signal* parameter is the signal number. The *Code* parameter is provided only for compatibility with other UNIX-compatible systems. The *Code* parameter value is always 0. The *SCP* parameter points to the **sigcontext** structure that is later used to restore the previous execution context of the process. The **sigcontext** structure is defined in the **signal.h** file.

  A new signal mask is calculated and installed for the duration of the signal-catching function (or until **sigprocmask** or**sigsuspend** subroutine is made). This mask is formed by joining the process-signal mask (the mask associated with the action for the signal being delivered) and the mask corresponding to the signal being delivered. The mask associated with the signal-catching function is not allowed to block those signals that cannot be ignored. This is enforced by the kernel without causing an error to be indicated. If and when the signal-catching function returns, the original signal mask is restored (modified by any **sigprocmask** calls that were made since the signal-catching function was called) and the receiving process resumes execution at the point it was interrupted.

  The signal-catching function can cause the process to resume in a different context by calling the **longjmp** subroutine. When the **longjmp** subroutine is called, the process leaves the signal stack, if it is currently on the stack, and restores the process signal mask to the state when the corresponding **setjmp** subroutine was made.

  Once an action is installed for a specific signal, it remains installed until another action is explicitly requested (by another call to the **sigaction** subroutine), or until one of the **exec** subroutines is called. An exception to this is when the **SA_OLDSTYLE** bit is set. In this case the action of a caught signal gets set to the **SIG_DFL** action before the signal-catching function for that signal is called.

  If a signal action is set to a pointer to a function while the signal is pending, the signal remains pending.

  The signal handler should not wait directly or indirectly on the input from a different thread in the form of a variable, pipe or anything similar. This will cause a deadlock in the case of a multithreaded application. As this will be a programmer initiated deadlock, the application will not handle it.

  When signal-catching functions are invoked asynchronously with process execution, the behavior of some of the functions defined by this standard is unspecified if they are called from a signal-

catching function. The following set of functions are reentrant with respect to signals; that is, applications can invoke them, without restriction, from signal-catching functions:

_exit
access
alarm
cfgetispeed
cfgetospeed
cfsetispeed
cfsetospeed
chdir
chmod
chown
close
creat
dup
dup2
exec
execle
execve
fcntl
fork
fpathconf
fstat
getegid
geteuid
getgid
getgroups
getpgrp
getpid
getppid
getuid
kill
link
lseek
mkdir
mkfifo
open
pathconf
pause
pipe
pread
pwrite
raise

read
readx
rename
rmdir
setgid
setpgid
setpgrp
setsid
setuid
sigaction
sigaddset
sigdelset
sigemptyset
sigismember
signal
sigpending
sigprocmask
sigsuspend
sleep
stat
statx
sysconf
tcdrain
tcflow
tcflush
tcgetattr
tcgetpgrp
tcsendbreak
tcsetattr
tcsetpgrp
time
times
umask
uname
unlink
ustat
utime
wait
waitpid
write

All other subroutines should not be called from signal-catching functions since their behavior is undefined.

oaction
Points to a **sigaction** structure in which the signal action data in effect at the time of the **sigaction** subroutine is returned.

invec
Points to a **sigvec** structure that describes the action to be taken upon receipt of the *signal* parameter signal.

outvec
Points to a **sigvec** structure in which the signal action data in effect at the time of the **sigvec** subroutine is returned.

action
Specifies the action associated with a signal.

Return Values

Upon successful completion, the **sigaction** subroutine returns a value of 0. Otherwise, a value of **SIG_ERR** is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **sigaction** subroutine is unsuccessful and no new signal handler is installed if one of the following occurs:

Item	Description
EFAULT	The *action* or *oaction* parameter points to a location outside of the allocated address space of the process.
EINVAL	The *signal* parameter is not a valid signal number.
EINVAL	An attempt was made to ignore or supply a handler for the**SIGKILL**, **SIGSTOP**, and **SIGCONT** signals.

sigaltstack Subroutine

Purpose

Allows a thread to define and examine the state of an alternate stack for signal handlers.

Library

(**libc.a**)

Syntax

#include <signal.h>

int sigaltstack(const stack_t *ss, stack_t *oss);

Description

The **sigaltstack** subroutine allows a thread to define and examine the state of an alternate stack for signal handlers. Signals that have been explicitly declared to execute on the alternate stack will be delivered on the alternate stack.

If *ss* is not null pointer, it points to a **stack_t** structure that specifies the alternate signal stack that will take effect upon return from **sigaltstack** subroutine. The **ss_flags** member specifies the new stack state. If it is set to **SS_DISABLE**, the stack is disabled and **ss_sp** and **ss_ssize** are ignored. Otherwise the stack will be enabled, and the **ss_sp** and **ss_size** members specify the new address and size of the stack.

The range of addresses starting at **ss_sp**, up to but not including **ss_sp + ss_size**, is available to the implementation for use as the stack.

If *oss* is not a null pointer, on successful completion it will point to a **stack_t** structure that specifies the alternate signal stack that was in effect prior to the **sigaltstack** subroutine. The **ss_sp** and **ss_size** members specify the address and size of the stack. The **ss_flags** member specifies the stack's state, and may contain one of the following values:

Item	Description
SS_ONSTACK	The process is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the process is executing or it fails. This flag must not be modified by processes.
SS_DISABLE	The alternate signal stack is currently disabled.

The value of **SIGSTKSZ** is a system default specifying the number of bytes that would be used to cover the usual case when manually allocating an alternate stack area. The value **MINSIGSTKSZ** is defined to be the minimum stack size for a signal handler. In computing an alternate stack size, a program should add that amount to its stack requirements to allow for the system implementation overhead.

After a successful call to one of the exec functions, there are no alternate stacks in the new process image.

Parameters

Item	Description
ss	A pointer to a **stack_t** structure specifying the alternate stack to use during signal handling.
oss	A pointer to a **stack_t** structure that will indicate the alternate stack currently in use.

Return Values

Upon successful completion, **sigaltstack** subroutine returns 0. Otherwise, it returns -1 and set **errno** to indicate the error.

Item	Description
-1	Not successful and the **errno** global variable is set to one of the following error codes.

Error Codes

Item	Description
EINVAL	The *ss* parameter is not a null pointer, and the **ss_flags** member pointed to by *ss* contains flags other that **SS_DISABLE**.
ENOMEM	The size of the alternate stack area is less than **MINSIGSTKSZ**.
EPERM	An attempt was made to modify an active stack.

sigemptyset, sigfillset, sigaddset, sigdelset, or sigismember Subroutine

Purpose

Creates and manipulates signal masks.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>
```

int sigemptyset (*Set*)
sigset_t ***Set*;

```
int sigfillset (Set)
sigset_t *Set;
```

int sigaddset (*Set*, *SignalNumber***)**
sigset_t ***Set*;
int *SignalNumber*;

```
int sigdelset (Set, SignalNumber)
sigset_t *Set;
int SignalNumber;
```

```
int sigismember (Set, SignalNumber)
sigset_t *Set;
int SignalNumber;
```

Description

The **sigemptyset**, **sigfillset**, **sigaddset**, **sigdelset**, and **sigismember** subroutines manipulate sets of signals. These functions operate on data objects addressable by the application, not on any set of signals known to the system, such as the set blocked from delivery to a process or the set pending for a process.

The **sigemptyset** subroutine initializes the signal set pointed to by the *Set* parameter such that all signals are excluded. The **sigfillset** subroutine initializes the signal set pointed to by the *Set* parameter such that all signals are included. A call to either the **sigfillset** or **sigemptyset** subroutine must be made at least once for each object of the **sigset_t** type prior to any other use of that object.

The **sigaddset** and **sigdelset** subroutines respectively add and delete the individual signal specified by the *SignalNumber* parameter from the signal set specified by the *Set* parameter. The **sigismember** subroutine tests whether the *SignalNumber* parameter is a member of the signal set pointed to by the *Set* parameter.

Parameters

Item	Description
Set	Specifies the signal set.
SignalNumber	Specifies the individual signal.

Examples

To generate and use a signal mask that blocks only the **SIGINT** signal from delivery, enter the following:

```
#include <signal.h>

int return_value;
sigset_t newset;
sigset_t *newset_p;
 . . .
newset_p = &newset;
sigemptyset(newset_p);
```

```
sigaddset(newset_p, SIGINT);
return_value = sigprocmask (SIG_SETMASK, newset_p, NULL);
```

Return Values

Upon successful completion, the **sigismember** subroutine returns a value of 1 if the specified signal is a member of the specified set, or the value of 0 if not. Upon successful completion, the other subroutines return a value of 0. For all the preceding subroutines, if an error is detected, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **sigfillset**, **sigdelset**, **sigismember**, and **sigaddset** subroutines are unsuccessful if the following is true:

Item	Description
EINVAL	The value of the *SignalNumber* parameter is not a valid signal number.

siginterrupt Subroutine

Purpose

Sets restart behavior with respect to signals and subroutines.

Library

Standard C Library (**libc.a**)

Syntax

```
int siginterrupt ( Signal,  Flag)
 int Signal, Flag;
```

Description

The **siginterrupt** subroutine is used to change the subroutine restart behavior when a subroutine is interrupted by the specified signal. If the flag is false (0), subroutines are restarted if they are interrupted by the specified signal and no data has been transferred yet.

If the flag is true (1), the restarting of subroutines is disabled. If a subroutine is interrupted by the specified signal and no data has been transferred, the subroutine will return a value of -1 with the **errno** global variable set to **EINTR**. Interrupted subroutines that have started transferring data return the amount of data actually transferred. Subroutine interrupt is the signal behavior found on 4.1 BSD and AT&T System V UNIX systems.

Note that the BSD signal-handling semantics are not altered in any other way. Most notably, signal handlers always remain installed until explicitly changed by a subsequent **sigaction** or **sigvec** call, and the signal mask operates as documented in the **sigaction** subroutine. Programs can switch between restartable and interruptible subroutine operations as often as desired in the running of a program.

Issuing a **siginterrupt** call during the running of a signal handler causes the new action to take place on the next signal caught.

Restart does not occur unless it is explicitly specified with the **sigaction** or **sigvec** subroutine in the **libc.a** library.

This subroutine uses an extension of the **sigvec** subroutine that is not available in the BSD 4.2; hence, it should not be used if compatibility with earlier versions is needed.

Parameters

Item	Description
Signal	Indicates the signal.
Flag	Indicates true or false.

Return Values

A value of 0 indicates that the call succeeded. A value of -1 indicates that the supplied signal number is not valid.

signbit Macro

Purpose

Tests the sign.

Syntax

```
#include <math.h>

int signbit (x)
real-floating x;
```

Description

The **signbit** macro determines whether the sign of its argument value is negative. NaNs, zeros, and infinities have a sign bit.

Parameters

Item	Description
x	Specifies the value to be tested.

Return Values

The **signbit** macro returns a nonzero value if the sign of its argument value is negative.

sigpending Subroutine

Purpose

Returns a set of signals that are blocked from delivery.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>
```

```
int sigpending ( Set)
sigset_t *Set;
```

Description

The **sigpending** subroutine stores a set of signals that are blocked from delivery and pending for the calling thread, in the space pointed to by the *Set* parameter.

Parameters

Item	Description
Set	Specifies the set of signals.

Return Values

Upon successful completion, the **sigpending** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **sigpending** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The input parameter is outside the user's address space.

sigprocmask, sigsetmask, or sigblock Subroutine

Purpose

Sets the current signal mask.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>

int sigprocmask ( How,  Set,  OSet)
int How;
const sigset_t *Set;
sigset *OSet;

int sigsetmask ( SignalMask)
int SignalMask;

int sigblock (SignalMask)
int SignalMask;
```

Description

Note: The **sigprocmask**, **sigsetmask**, and **sigblock** subroutines must not be used in a multi-threaded application. The **sigthreadmask** (“sigthreadmask Subroutine” on page 1962) subroutine must be used instead.

The **sigprocmask** subroutine is used to examine or change the signal mask of the calling thread.

The subroutine is used to examine or change the signal mask of the calling process.

Typically, you should use the **sigprocmask(SIG_BLOCK)** subroutine to block signals during a critical section of code. Then use the **sigprocmask(SIG_SETMASK)** subroutine to restore the mask to the previous value returned by the **sigprocmask(SIG_BLOCK)** subroutine.

If there are any pending unblocked signals after the call to the **sigprocmask** subroutine, at least one of those signals will be delivered before the **sigprocmask** subroutine returns.

The **sigprocmask** subroutine does not allow the **SIGKILL** or **SIGSTOP** signal to be blocked. If a program attempts to block either signal, the **sigprocmask** subroutine gives no indication of the error.

Parameters

Item	Description
How	Indicates the manner in which the set is changed. It can have one of the following values: **SIG_BLOCK** The resulting set is the union of the current set and the signal set pointed to by the *Set* parameter. **SIG_UNBLOCK** The resulting set is the intersection of the current set and the complement of the signal set pointed to by the *Set* parameter. **SIG_SETMASK** The resulting set is the signal set pointed to by the *Set* parameter.
Set	Specifies the signal set. If the value of the *Set* parameter is not null, it points to a set of signals to be used to change the currently blocked set. If the value of the *Set* parameter is null, the value of the *How* parameter is not significant and the process signal mask is unchanged. Thus, the call can be used to inquire about currently blocked signals.
OSet	If the *OSet* parameter is not the null value, the signal mask in effect at the time of the call is stored in the space pointed to by the *OSet* parameter.
SignalMask	Specifies the signal mask of the process.

Compatibility Interfaces

The **sigsetmask** subroutine allows changing the process signal mask for signal values 1 to 31. This same function can be accomplished for all values with the **sigprocmask(SIG_SETMASK)** subroutine. The signal of value *i* will be blocked if the *i*th bit of *SignalMask* parameter is set.

Upon successful completion, the **sigsetmask** subroutine returns the value of the previous signal mask. If the subroutine fails, a value of -1 is returned and the **errno** global variable is set to indicate the error as in the **sigprocmask** subroutine.

The **sigblock** subroutine allows signals with values 1 to 31 to be logically ORed into the current process signal mask. This same function can be accomplished for all values with the **sigprocmask(SIG_BLOCK)** subroutine. The signal of value *i* will be blocked, in addition to those currently blocked, if the *i*-th bit of the *SignalMask* parameter is set.

It is not possible to block a **SIGKILL** or **SIGSTOP** signal using the **sigblock** or **sigsetmask** subroutine. This restriction is *silently* imposed by the system without causing an error to be indicated.

Upon successful completion, the **sigblock** subroutine returns the value of the previous signal mask. If the subroutine fails, a value of -1 is returned and the **errno** global variable is set to indicate the error as in the **sigprocmask** subroutine.

Return Values

Upon completion, a value of 0 is returned. If the **sigprocmask** subroutine fails, the signal mask of the process is unchanged, a value of -1 is returned, and the global variable **errno** is set to indicate the error.

Error Codes

The **sigprocmask** subroutine is unsuccessful if the following is true:

Item	Description
EPERM	The user does not have the privilege to change the signal's mask.
EINVAL	The value of the *How* parameter is not equal to one of the defined values.
EFAULT	The user's mask is not in the process address space.

Examples

To set the signal mask to block only the **SIGINT** signal from delivery, enter:

```
#include <signal.h>

int return_value;
sigset_t newset;
sigset_t *newset_p;
 . . .
newset_p = &newset;
sigemptyset(newset_p);
sigaddset(newset_p, SIGINT);
return_value = sigprocmask (SIG_SETMASK, newset_p, NULL);
```

sigqueue Subroutine

Purpose

Queues a signal to a process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>

int sigqueue (pid, signo, value)
pid_t pid;
int signo;
const union sigval value;
```

Description

The **sigqueue** subroutine causes the signal specified by the *signo* parameter to be sent with the value specified by the *value* parameter to the process specified by the *pid* parameter. If the *signo* parameter is zero, error checking is performed but no signal is actually sent. This can be used to check the validity of the *pid* parameter.

The conditions required for a process to have permission to queue a signal to another process are the same as for the **kill** subroutine.

The **sigqueue** subroutine returns immediately. If **SA_SIGINFO** is set by the receiving process for the specified signal, and if the resources are available to queue the signal, the signal is queued and sent to the receiving process. If **SA_SIGINFO** is not set for the *signo* parameter, the signal is sent at least once to the receiving process.

If multiple signals in the range **SIGRTMIN** to **SIGRTMAX** should be available for delivery, the lowest numbered of them will be delivered first.

Parameters

Item	Description
pid	Specifies the process to which a signal is to be sent.
signo	Specifies the signal number.
value	Specifies the value to be sent with the signal.

Return Values

Upon successful completion the **sigqueue** subroutine returns a zero. If unsuccessful, it returns a -1 and sets the **errno** variable to indicate the error.

Error Code

The **sigqueue** subroutine will fail if:

Item	Description
EAGAIN	No resources are available to queue the signal. The process has already queued **SIGQUEUE_MAX** signals that are still pending at the receiver(s), or a system-wide resource limit has been exceeded.
EINVAL	The value of the *signo* parameter is an invalid or unsupported signal number, or if the selected signal can either stop or continue the receiving process. AIX does not support queuing of the following signals: SIGKILL, SIGSTOP, SIGTSTP, SIGCONT, SIGTTIN, SIGTTOU, and SIGCLD.
EPERM	The process does not have the appropriate privilege to send the signal to the receiving process.
ESRCH	The process specified by the *pid* parameter does not exist.

sigset, sighold, sigrelse, or sigignore Subroutine

Purpose

Enhance the signal facility and provide signal management.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>
void (*sigset( Signal,  Function))()
int Signal;
void (*Function)();
int sighold ( Signal)
int Signal;
```

```
int sigrelse ( Signal)
int Signal;
int sigignore ( Signal)
int Signal;
```

Description

The **sigset**, **sighold**, **sigrelse**, and **sigignore** subroutines enhance the signal facility and provide signal management for application processes.

The **sigset** subroutine specifies the system signal action to be taken upon receiving a *Signal* parameter.

The **sighld** and **sigrelse** subroutines establish critical regions of code. A call to the **sighold** subroutine is analogous to raising the priority level and deferring or holding a signal until the priority is lowered by **sigrelse**. The **sigrelse** subroutine restores the system signal action to the action that was previously specified by the **sigset** structure.

The **sigignore** subroutine sets the action for the *Signal* parameter to **SIG_IGN**.

The other signal management routine, **signal**, should not be used in conjunction with these routines for a particular signal type.

Parameters

Item	Description
Signal	Specifies the signal. The *Signal* parameter can be assigned any one of the following signals: **SIGHUP** Hang up **SIGINT** Interrupt **SIGQUIT** Quit* **SIGILL** Illegal instruction (not reset when caught)* **SIGTRAP** Trace trap (not reset when caught)* **SIGABRT** Abort* **SIGFPE** Floating point exception*, or arithmetic exception, integer divide by 0 **SIGSYS** Bad argument to routine* **SIGPIPE** Write on a pipe with no one to read it **SIGALRM** Alarm clock **SIGTERM** Software termination signal **SIGUSR1** User-defined signal 1 **SIGUSR2** User-defined signal 2. * The default action for these signals is an abnormal termination.

For portability, application programs should use or catch only the signals listed above. Other signals are hardware-dependant and implementation-dependant and may have very different meanings or results across systems. For example, the System V signals (**SIGEMT**, **SIGBUS**, **SIGSEGV**, and **SIGIOT**) are implementation-dependent and are not listed above. Specific implementations may have other implementation-dependent signals.

Item	Description
Function	Specifies the choice. The *Function* parameter is declared as a type pointer to a function returning void. The *Function* parameter is assigned one of four values: **SIG_DFL**, **SIG_IGN**, **SIG_HOLD**, or an *address* of a signal-catching function. Definitions of the actions taken by each of the values are: **SIG_DFL** Terminate process upon receipt of a signal. Upon receipt of the signal specified by the *Signal* parameter, the receiving process is to be terminated with all of the consequences outlined in the **_exit** subroutine. In addition, if *Signal* is one of the signals marked with an asterisk above, implementation-dependent abnormal process termination routines, such as a core dump, can be invoked. **SIG_IGN** Ignore signal. Any pending signal specified by the *Signal* parameter is discarded. A pending signal is a signal that has occurred but for which no action has been taken. The system signal action is set to ignore future occurrences of this signal type. **SIG_HOLD** Hold signal. The signal specified by the *Signal* parameter is to be held. Any pending signal of this type remains held. Only one signal of each type is held.
address	Catch signal. Upon receipt of the signal specified by the *Signal* parameter, the receiving process is to execute the signal-catching function pointed to by the *Function* parameter. Any pending signal of this type is released. This address is retained across calls to the other signal management functions, **sighold** and **sigrelse**. The signal number *Signal* is passed as the only argument to the signal-catching function. Before entering the signal-catching function, the value of the *Function* parameter for the caught signal is set to **SIG_HOLD**. During normal return from the signal-catching handler, the system signal action is restored to the *Function* parameter and any held signal of this type is released. If a nonlocal goto (see the **setjmp** subroutine) is taken, the **sigrelse** subroutine must be invoked to restore the system signal action and to release any held signal of this type. Upon return from the signal-catching function, the receiving process will resume execution at the point at which it was interrupted, except for implementation-defined signals in which this may not be true. When a signal to be caught occurs during a nonatomic operation such as a call to the **read**, **write**, **open**, or **ioctl** subroutine on a slow device (such as a terminal); during a **pause** subroutine; during a **wait** subroutine that does not return immediately, the signal-catching function is executed. The interrupted routine then returns a value of -1 to the calling process with the **errno** global variable set to **EINTR**.

Return Values

Upon successful completion, the **sigset** subroutine returns the previous value of the system signal action for the specified *Signal.* Otherwise, it returns **SIG_ERR** and the **errno** global variable is set to indicate the error.

For the **sighold**, **sigrelse**, and **sigignore** subroutines, a value of 0 is returned upon success. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **sigset**, **sighold**, **sigrelse**, or **sigignore** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *Signal* value is either an illegal signal number, or the default handling of *Signal* cannot be changed.

sigsetjmp or siglongjmp Subroutine

Purpose

Saves or restores stack context and signal mask.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <setjmp.h>

int sigsetjmp ( Environment,  SaveMask)
sigjmp_buf Environment;
int SaveMask;

void siglongjmp (Environment,  Value)
sigjmp_buf Environment;
int Value;
```

Description

The **sigsetjmp** subroutine saves the current stack context, and if the value of the *SaveMask* parameter is not 0, the **sigsetjmp** subroutine also saves the current signal mask of the process as part of the calling environment.

The **siglongjmp** subroutine restores the saved signal mask only if the *Environment* parameter was initialized by a call to the **sigsetjmp** subroutine with a nonzero *SaveMask* parameter argument.

Parameters

Item	Description
Environment	Specifies an address for a **sigjmp_buf** structure.
SaveMask	Specifies the flag used to determine if the signal mask is to be saved.
Value	Specifies the return value from the **siglongjmp** subroutine.

Return Values

The **sigsetjmp** subroutine returns a value of 0. The **siglongjmp** subroutine returns a nonzero value.

sigstack Subroutine

Purpose

Sets and gets signal stack context.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>

int sigstack ( InStack,  OutStack)
struct sigstack *InStack, *OutStack;
```

Description

The **sigstack** subroutine defines an alternate stack on which signals are to be processed.

When a signal occurs and its handler is to run on the signal stack, the system checks to see if the process is already running on that stack. If so, it continues to do so even after the handler returns. If not, the signal handler runs on the signal stack, and the original stack is restored when the handler returns.

Use the **sigvec** or **sigaction** subroutine to specify whether a given signal-handler routine is to run on the signal stack.

Attention: A signal stack does not automatically increase in size as a normal stack does. If the stack overflows, unpredictable results can occur.

Parameters

Item	Description

InStack	Specifies the stack pointer of the new signal stack.
	If the value of the *InStack* parameter is nonzero, it points to a **sigstack** structure, which has the following members:
	`caddr_t   ss_sp;`
	`int   ss_onstack;`
	The value of `InStack->ss_sp` specifies the stack pointer of the new signal stack. Since stacks grow from numerically greater addresses to lower ones, the stack pointer passed to the **sigstack** subroutine should point to the numerically high end of the stack area to be used. `InStack->ss_onstack` should be set to a value of 1 if the process is currently running on that stack; otherwise, it should be a value of 0.
	If the value of the *InStack* parameter is 0 (that is, a null pointer), the signal stack state is not set.

OutStack	Points to structure where current signal stack state is stored. If the value of the *OutStack* parameter is nonzero, it points to a **sigstack** structure into which the **sigstack** subroutine stores the current signal stack state. If the value of the *OutStack* parameter is 0, the previous signal stack state is not reported.

Return Values

Upon successful completion, the **sigstack** subroutine returns a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **sigstack** subroutine is unsuccessful and the signal stack context remains unchanged if the following is true:

Item	Description
EFAULT	The *InStack* or *OutStack* parameter points outside of the address space of the process.

sigsuspend or sigpause Subroutine

Purpose

Automatically changes the set of blocked signals and waits for a signal.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>

int sigsuspend ( SignalMask)
const sigset_t *SignalMask;
```

```
int sigpause (SignalMask)
int SignalMask;
```

Description

The **sigsuspend** subroutine replaces the signal mask of a thread with the set of signals pointed to by the *SignalMask* parameter. It then suspends execution of the thread until a signal is delivered that executes a signal-catching function or terminates the process. The **sigsuspend** subroutine does not allow the **SIGKILL** or **SIGSTOP** signal to be blocked. If a program attempts to block one of these signals, the **sigsuspend** subroutine gives no indication of the error.

If delivery of a signal causes the process to end, the **sigsuspend** subroutine does not return. If delivery of a signal causes a signal-catching function to start, the **sigsuspend** subroutine returns after the signal-catching function returns, with the signal mask restored to the set that existed prior to the **sigsuspend** subroutine.

The **sigsuspend** subroutine sets the signal mask and waits for an unblocked signal as one atomic operation. This means that signals cannot occur between the operations of setting the mask and waiting for a signal. If a program invokes the **sigprocmask (SIG_SETMASK)** and **pause** subroutines separately, a signal that occurs between these subroutines might not be noticed by the **pause** subroutine.

In normal usage, a signal is blocked by using the **sigprocmask(SIG_BLOCK,...)** subroutine for single-threaded applications, or the **sigthreadmask(SIG_BLOCK,...)** subroutine for multi-threaded applications (using the **libpthreads.a** threads library) at the beginning of a critical section. The process/thread then determines whether there is work for it to do. If no work is to be done, the process/thread waits for work by calling the **sigsuspend** subroutine with the mask previously returned by the **sigprocmask** or **sigthreadmask** subroutine.

The **sigpause** subroutine is provided for compatibility with older UNIX systems; its function is a subset of the **sigsuspend** subroutine.

Parameter

Item	Description
SignalMask	Points to a set of signals.

Return Values

If a signal is caught by the calling thread and control is returned from the signal handler, the calling thread resumes execution after the **sigsuspend** or **sigpause** subroutine, which always return a value of -1 and set the **errno** global variable to **EINTR.**

sigthreadmask Subroutine

Purpose

Sets the signal mask of a thread.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include <pthread.h>
#include <signal.h>

int sigthreadmask( how,  set,  old_set)
int how;
const sigset_t *set;
sigset_t *old_set;
```

Description

The **sigthreadmask** subroutine is used to examine or change the signal mask of the calling thread. The **sigprocmask** subroutine must not be used in a multi-threaded process.

Typically, the **sigthreadmask(SIG_BLOCK)** subroutine is used to block signals during a critical section of code. The **sigthreadmask(SIG_SETMASK)** subroutine is then used to restore the mask to the previous value returned by the **sigthreadmask(SIG_BLOCK)** subroutine.

If there are any pending unblocked signals after the call to the **sigthreadmask** subroutine, at least one of those signals will be delivered before the **sigthreadmask** subroutine returns.

The **sigthreadmask** subroutine does not allow the **SIGKILL** or **SIGSTOP** signal to be blocked. If a program attempts to block either signal, the **sigthreadmask** subroutine gives no indication of the error.

Note: The **pthread.h** header file must be the first included file of each source file using the threads library.

Parameters

Item	Description
how	Indicates the manner in which the set is changed. It can have one of the following values: **SIG_BLOCK** The resulting set is the union of the current set and the signal set pointed to by the *set* parameter. **SIG_UNBLOCK** The resulting set is the intersection of the current set and the complement of the signal set pointed to by the *set* parameter. **SIG_SETMASK** The resulting set is the signal set pointed to by the *set* parameter.
set	Specifies the signal set. If the value of the *Set* parameter is not null, it points to a set of signals to be used to change the currently blocked set. If the value of the *Set* parameter is null, the value of the *How* parameter is not significant and the process signal mask is unchanged. Thus, the call can be used to inquire about currently blocked signals.
old_set	If the *old_set* parameter is not the null value, the signal mask in effect at the time of the call is stored in the spaced pointed to by the *old_set* parameter.

Return Values

Upon completion, a value of 0 is returned. If the **sigthreadmask** subroutine fails, the signal mask of the process is unchanged, a value of -1 is returned, and the global variable **errno** is set to indicate the error.

Error Codes

The **sigthreadmask** subroutine is unsuccessful if the following is true:

Item	Description
EFAULT	The *set* or *old_set* pointers are not in the process address space.
EINVAL	The value of the *how* parameter is not supported.
EPERM	The calling thread does not have the privilege to change the signal's mask.

Examples

To set the signal mask to block only the **SIGINT** signal from delivery, enter:

```
#include <pthread.h>
#include <signal.h>

int return_value;
sigset_t newset;
sigset_t *newset_p;
 . . .
newset_p = &newset;
sigemptyset(newset_p);
sigaddset(newset_p, SIGINT);
return_value = sigthreadmask(SIG_SETMASK, newset_p, NULL);
```

sigtimedwait and sigwaitinfo Subroutine

Purpose

Waits for a signal, and provides a mechanism for retrieving any queued value.

Library

Standard C Library (**libc.a**)

Threads Library (**libpthreads.a**)

Syntax

```
#include <signal.h>

int sigtimedwait (set, info, timeout)
const sigset_t *set;
siginfo_t *info;
const struct timespec *timeout;

int sigwaitinfo (set, info)
const sigset_t *set;
siginfo_t *info;
```

Description

The **sigwaitinfo** subroutine selects a pending signal from the set specified by the *set* parameter. If no signal in the *set* parameter is pending at the time of the call, the calling thread is suspended until one or more signals in the *set* parameter become pending or until it is interrupted by an unblocked, caught signal. If the wait was interrupted by an unblocked, caught signal, the subroutines will restart themselves.

The **sigwaitinfo** subroutine is functionally equivalent to the **sigwait** subroutine if the *info* argument is NULL. If the *info* argument is non-NULL, the **sigwaitinfo** subroutine is equivalent to the **sigwait** subroutine, except that the selected signal number is stored in the **si_signo** member, and the cause of the signal is stored in the **si_code** member of the *info* parameter. If any value is queued to the selected signal, the first such queued value is dequeued, and if the *info* argument is non-NULL, the value is stored in the **si_value** member of the *info* parameter. If no further signals are queued for the selected signal, the pending indication for that signal is reset.

The **sigtimedwait** subroutine is equivalent to the **sigwaitinfo** subroutine except that if none of the signals specified by the *set* parameter are pending, the **sigtimedwait** subroutine waits for the time interval referenced by the *timeout* parameter. If the **timespec** structure pointed to by the *timeout* parameter contains a zero value and if none of the signals specified by the *set* parameter are pending, the **sigtimedwait** subroutine returns immediately with an error.

If there are multiple pending signals in the range **SIGRTMIN** to **SIGRTMAX**, the lowest numbered signal in that range will be selected.

Note: All signals in set should have been blocked prior to calling any of the **sigwait** subroutines.

Parameters

Item	Description
set	Specifies the pending signals that may be selected.
info	Points to a **siginfo_t** in which additional signal information can be returned.
timeout	Points to the **timespec** structure.

Return Values

Upon successful completion, the **sigtimedwait** and **sigwaitinfo** subroutines return the selected signal number. If unsuccessful, the **sigtimedwait** and **sigwaitinfo** subroutines return -1 and set the **errno** variable to indicate the error.

Error Codes

The **sigtimedwait** subroutine will fail if:

Item	Description
EAGAIN	No signal specified by the *set* parameter was generated within the specified timeout period.

The **sigtimedwait** and **sigwaitinfo** subroutines may fail if:

Item	Description
EINVAL	The *set* parameter is empty, or contains an invalid, non-catchable, or unsupported signal number.

The **sigtimedwait** subroutine may also fail when none of the selected signals are pending if:

Item	Description
EINVAL	The *timeout* parameter specified a tv_nsec value less than zero or greater than or equal to 1000 million.

sigwait Subroutine

Purpose

Blocks the calling thread until a specified signal is received.

Library

Threads Library (**libpthreads.a**)

Syntax

```
#include </usr/include/sys/signal.h>

int sigwait ( set,  sig)
const sigset_t *set;
int *sig;
```

Description

The **sigwait** subroutine blocks the calling thread until one of the signal in the signal set *set* is received by the thread. **sigwait** returns an **EINVAL** error if it attempts to wait on **SIGKILL(9)**, **SIGSTOP(17)**, or **SIGWAITING(39**–AIX-specific**)**.

The signal can be either sent directly to the thread, using the **pthread_kill** subroutine, or to the process. In that case, the signal will be delivered to exactly one thread that has not blocked the signal.

Concurrent use of **sigaction** and **sigwait** subroutines on the same signal is forbidden.

Parameters

Item	Description
set	Specifies the set of signals to wait on.
sig	Points to where the received signal number will be stored.

Return Values

Upon successful completion, the received signal number is returned via the *sig* parameter, and 0 is returned. Otherwise, an error code is returned.

Error Code

The **sigwait** subroutine is unsuccessful if the following is true:

Item	Description
EINVAL	The *set* parameter contains an invalid or unsupported signal number.

sin, sinf, sinl, sind32, sind64, and sind128 Subroutine

Purpose

Computes the sine.

Syntax

```
#include <math.h>
```

double sin (*x*)
double *x*;

```
float sinf (x)
float x;
```

```
long double sinl (x)
long double x;
```

```
_Decimal32 sind32 (x)
_Decimal32 x;

_Decimal64 sind64 (x)
_Decimal64 x;

_Decimal128 sind128 (x)
_Decimal128 x;
```

Description

The **sin**, **sinf**, **sinl**, **sind32**, **sind64**, and **sind128** subroutines compute the sine of the *x* parameter, measured in radians.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Floating-point value
y	Floating-point value

Return Values

Upon successful completion, the **sin**, **sinf**, **sinl**, **sind32**, **sind64**, and **sind128** subroutines return the sine of *x*.

If *x* is NaN, a NaN is returned.

If *x* is ±0, *x* is returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

If *x* is ±Inf, a domain error occurs, and a NaN is returned.

Error Codes

The **sin**, **sinf**, and **sinl** subroutines lose accuracy when passed a large value for the *x* parameter. In the **sin** subroutine, for example, values of *x* that are greater than pi are argument-reduced by first dividing them by the machine value for 2 * `pi` , and then using the IEEE remainder of this division in place of *x.* Since the machine value of `pi` can only approximate its infinitely precise value, the remainder of x/(2 * `pi`) becomes less accurate as *x* becomes larger. Similar loss of accuracy occurs for the **sinl** subroutine during argument reduction of large arguments.

Item	Description
sin	When the *x* parameter is extremely large, these functions return 0 when there would be a complete loss of significance. In this case, a message indicating **TLOSS** error is printed on the standard error output. For less extreme values causing partial loss of significance, a **PLOSS** error is generated but no message is printed. In both cases, the **errno** global variable is set to a **ERANGE** value.

These error-handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** (**-lmsaa**) library.

sinh, sinhf, sinhl, sinhd32, sinhd64, and sinhd128 Subroutines

Purpose

Computes hyperbolic sine.

Syntax

```
#include <math.h>
```

```
double sinh ( x)
double x;
```

```
float sinhf (x)
float x;
```

```
long double sinhl (x)
long double x;

_Decimal32 sinhd32 (x)
_Decimal32 x;

_Decimal64 sinhd64 (x)
_Decimal64 x;
```

```
_Decimal128 sinhd128 (x)
_Decimal128 x;
```

Description

The **sinh**, **sinhf**, **sinhl**, **sinhd32**, **sinhd64**, and **sinhd128** subroutines compute the hyperbolic sine of the *x* parameter.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies a double-precision floating-point value.

Return Values

Upon successful completion, the **sinh**, **sinhf**, **sinhl**, **sinhd32**, **sinhd64**, and **sinhd128** subroutines return the hyperbolic sine of *x*.

If the result would cause an overflow, a range error occurs and ±**HUGE_VAL**, ±**HUGE_VALF**, ±**HUGE_VALL**, ±**HUGE_VAL_D32**, ±**HUGE_VAL_D64**, and ±**HUGE_VAL_D128** (with the same sign as *x*) is returned as appropriate for the type of the function.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or infinite, *x* is returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

Error Codes

If the correct value overflows, the **sinh**, **sinhf**, **sinhl**, **sinhd32**, **sinhd64**, and **sinhd128** subroutines return a correctly signed **HUGE_VAL**, and the **errno** global variable is set to **ERANGE**.

These error-handling procedures should be changed with the **matherr** subroutine when the **libmsaa.a** (**-lmsaa**) library is used.

sl_clr or tl_clr Subroutine

Purpose

Resets the labels.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>
```

```
int sl_clr (sl)
sl_t *sl;
```

```
int tl_clr (tl)
tl_t *tl;
```

Description

The **sl_clr** and **tl_clr** subroutines reset the labels. These subroutines set any content in the label structure to zero.

Parameters

Item	Description
sl	Points to the sensitivity label to be cleared.
tl	Points to the integrity label to be cleared.

Return Values

Item	Description
0	Indicates a successful completion.
1	Indicates that an error occurred.

Error Codes

Item	Description
EINVAL	Indicates that the passed-in parameter is NULL.

sl_cmp or tl_cmp Subroutine

Purpose

Compares sensitivity and integrity labels.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>

CMP_RES_T sl_cmp (sl1, sl2)
const sl_t *sl1;
const sl_t *sl2;

CMP_RES_T tl_cmp (tl1, tl2)
const tl_t *tl1;
const tl_t *tl2;
```

Description

The **sl_cmp** and **tl_cmp** subroutines compare two labels. There are three types of relationship between labels: dominance, equality, and non-comparable.

Sensitivity label (SL) comparison is made based on the following conditions:

Dominance:

One SL (L1) dominates another (L2) if and only if the L1 meets the following requirement:

- The classification in L1 equals or exceeds the classification in L2.
- The set of compartments in L1 completely contains the set of compartments in L2.

Equality:

One SL (L1) equals another SL (L2) if and only if the L1 meets the following requirement:

- The classification in L1 equals the classification in L2.
- The set of compartments in L1 is identical to the set of compartments in L2.

Non-comparable:

Two labels can be disjoint (L1 is not equal to L2, and L1 does not dominate L2, and L2 does not dominate L1). One SL (L1) is non-comparable to another (L2) if the L1 meets the following requirement:

- The set of compartments in L1 does not completely contain the set in L2 and L2 does not completely contain the set in L1.

Therefore, they are considered disjoint.

Integrity label (TL) comparison is made based on the following conditions:

Dominance:

One TL (L1) dominates another (L2) if and only if the L1 meets the following requirement:

- The classification in L1 equals or exceeds the classification in L2.

Equality:

One TL (L1) equals another SL (L2) if and only if the L1 meets the following requirement:

- The classification in L1 equals the classification in L2.

Parameters

Item	Description
sl1, sl2	Specifies sensitivity labels to be compared.
tl1, tl2	Specifies Integrity labels to be compared.

Return Values

Item	Description
LAB_DOM	Indicates that sl1 dominates sl2.
LAB_SAME	Indicates that sl1 is identical to sl2.
LAB_IDOM	Indicates that sl2 dominates sl1.
LAB_NCMP	Indicates that sl1 and sl2 are non-comparable.
LAB_ERR	Indicates that the parameter is not valid.

Note: For the **tl_cmp** subroutine, if either of the integrity labels passed evaluates to the special TL NOTL, the subroutine returns the **LAB_DOM** value.

Error Codes

Item	Description
EINVAL	Indicates that the passed-in parameter is NULL.

slbtohr, slhrtob, clbtohr, clhrtob, tlbtohr, or tlhrtob Subroutine

Purpose

Converts labels from binary equivalent to human readable format and from human readable format to binary equivalent.

Library

Trusted AIX Library (**libmls.a**)

Syntax

```
#include <mls/mls.h>
```

```
int slbtohr (hr_sl, sl, type)
char *hr_sl;
const sl_t *sl;
enum hr_type type;

int clbtohr (hr_cl, cl, type)
char *hr_cl;
const sl_t *cl;
enum hr_type type;

int tlbtohr (hr_tl, tl, type)
char *hr_tl;
const tl_t *tl;
enum hr_type type;

int clhrtob (cl, hr_cl)
sl_t *cl;
const char *hr_cl;

int slhrtob (sl, hr_sl)
sl_t *sl;
const char *hr_sl;

int tlhrtob (tl, hr_tl)
tl_t *tl;
const char *hr_tl;
```

Description

The **btohr** routines convert the binary labels into long or short human readable form, based on the value of the *type* parameter.

The **slbtohr** subroutine converts binary sensitivity labels to human readable form, that is, the conversion is made as per SENSITIVITY LABELS section of Label Encoding File.

The **clbtohr** subroutine converts binary clearance labels to human readable form, that is, the conversion is made as per as per CLEARANCE LABELS section of Label Encoding File.

The **tlbtohr** subroutine converts binary integrity labels to human readable form, that is, the conversion is made as per optional INTEGRITY LABELS or SENSITIVITY LABELS section of Label Encoding File.

Similarly, the respective **hrtob** routines convert human (short or long) readable form to binary format.

Note: The database has to be initialized before you start any of these routines.

Parameters

The **btohr** routines have the following parameters:

Item	Description
hr_sl	Points to the human readable forms of binary labels. This buffer is expected to be of length determined by the **maxlen_sl** subroutine.
hr_cl	Points to the human readable forms of binary labels. This buffer is expected to be of length determined by the **maxlen_cl** subroutine.
hr_tl	Points to the human readable forms of binary labels. This buffer is expected to be of length determined by the **maxlen_tl** subroutine.
sl	Points to the binary sensitivity label of **sl_t *** type.
cl	Points to the clearance label of **sl_t *** type.
tl	Points to the integrity label of **tl_t *** type.
type	Specifies the human readable format the binary label is to be converted to. It can be one of the following values: **HR_LONG** Specifies the long human readable format. **HR_SHORT** Specifies the short human readable format.

The **hrtob** routines have the following parameters:

Item	Description
hr_sl	Points to the human readable labels, either short form or long form.
hr_cl	Points to the human readable labels, either short form or long form.
hr_tl	Points to the human readable labels, either short form or long form.
sl	Points to binary sensitivity labels.
cl	Points to clearance labels.
tl	Points to binary integrity label.

Security

Files Accessed:

Modes	File
R	/etc/security/enc/LabelEncodings

Return Values

Item	Description
0	Indicates a successful completion.
1	Indicates that an error occurred.

Error Codes

Item	Description
EINVAL	Indicates that the passed-in parameter is NULL.
ENOTREADY	Indicates that the database is not initialized.

sleep, nsleep or usleep Subroutine

Purpose

Suspends a current process from execution.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
unsigned int sleep ( Seconds)

#include <sys/time.h>
int nsleep ( Rqtp,  Rmtp)
struct timestruc_t *Rqtp, *Rmtp;

int usleep ( Useconds)
useconds_t Useconds;
```

Description

The **sleep**, **usleep**, or **nsleep** subroutines suspend the current process until:

- The time interval specified by the *Seconds*, *Useconds*, or *Rqtp* parameter elapses.
- A signal is delivered to the calling process that starts a signal-catching function or end the process.
- The process is notified of an event through an event notification function.

The suspension time might be longer than requested time due to the scheduling of other activity by the system. Upon return, the location that is specified by the *Rmtp* parameter is updated to show the time that is left in the interval, or 0 if the full interval is elapsed.

Parameters

Item	Description
Rqtp	Time interval specified for suspension of execution.
Rmtp	Specifies the time is left on the interval timer or 0.
Seconds	Specifies time interval in seconds.
Useconds	Specifies time interval in microseconds. This parameter is available only for the **usleep** subroutine.

Compatibility Interfaces

The **sleep** and **usleep** subroutines are simplified forms for the **nsleep** subroutine. These subroutines ensure compatibility with older versions of the Portable Operating System Interface (POSIX) and Linux® specifications. The **sleep** subroutine suspends the current process for whole seconds. The **usleep**

subroutine suspends the current process in microseconds, and the **nsleep** subroutine suspends the current process in nanoseconds.

In AIX Version 5.1, or later, time is measured in nanoseconds. The **nsleep** subroutine is the system call that is used by the AIX operating system to suspend thread execution. The **sleep** and **usleep** subroutines serve as front end to the **nsleep** subroutine.

The actual time interval for which the process is suspended is approximate. The time interval to suspend a process might take long time because of the other activities that are scheduled by the system, or the process suspension might take less time because of a signal that preempts the suspension.

For the **nsleep** subroutine, your must specify the *Rqtp* (Requested Time Pause) and *Rmtp* (Remaining Time Pause) parameters so that the actual time for which the process is suspended can be identified. Normally, the value in *Rmtp* parameter is the equivalent of zero. By design, the maximum value that might be used in the *Rqtp* parameter is the number of nanoseconds in one second.

Example

To suspend a current running process for 10 seconds, enter the following command:

```
sleep (10)
```

Return Values

The **nsleep**, **sleep**, and **usleep** subroutines return a value of 0 if the requested time is elapsed.

If the **nsleep** subroutine returns a value of -1, the notification of a signal or event was received and the *Rmtp* parameter is updated to the requested time minus the time slept (unslept time), and the **errno** global variable is set.

If the **sleep** subroutine returns because of a premature arousal due to delivery of a signal, the return value is the unslept amount (the requested time minus the time slept) in seconds.

Error Codes

If the **nsleep** subroutine fails, a value of -1 is returned and the **errno** global variable is set to one of the following error codes:

Item	Description
EINTR	A signal was detected by the calling process and control is returned from the signal-catching routine, or the process is notified of an event through an event notification function.
EINVAL	The *Rqtp* parameter specified a nanosecond value less than zero or greater than or equal to 1 second.
EFAULT	An argument address referenced informed memory. **Note:** An **errno** can be set to **EFAULT** as well.

The **sleep** subroutine is always successful and no return value is reserved to indicate an error.

slk_attroff, slk_attr_off, slk_attron, slk_attrset, slk_attr_set, slk_clear, slk_color, slk_init, slk_label, slk_noutrefresh, slk_refresh, slk_restore, slk_set, slk_touch, slk_wset, Subroutine

Purpose

Soft label subroutines.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int slk_attroff
(const chtype attrs);

int slk_attr_off
(const attr_t attrs,
void *opts);

int slk_attron
(const chtype attrs);

int slk_attr_on
(const attr_t attrs,
void *opts);

int slk_attrset
(const chtype attrs);

int slk_attr_set
(const attr_t attrs,
short color_pair_number,
void *opts);

int slk_clear
(void);

int slk_color
(short color_pair_number);

int slk_init
(int fmt);

char *slk_label
(int labnum);

int slk_noutrefresh
(void);

int slk_refresh
(void);

int slk_restore
(void);

int slk_set
(int labnum,
const char *label,
int justify);

int slk_touch
(void);

int slk_wset
(int labnum,
const wchar_t *label,
int justify);
```

Description

The Curses interface manipulates the set of soft function-key labels that exist on many terminals. For those terminals that do not have sort labels, Curses takes over the bottom line of *stdscr*, reducing the size of *stdscr* and the value of the LINES external variable. There can be up to eight labels of up to eight display columns each.

To use soft labels, the **slk_init** subroutine must be called before **initscr**, **newterm**, or **ripoffline** is called. If **initscr** eventually uses a line from *stdscr* to emulate the soft labels, then *fmt* determines how the labels

are arranged on the screen. Setting *fmt* to **0** indicates a 3-2-3 arrangement of the labels; **1** indicates a 4-4 arrangement. Other values for *fmt* are unspecified.

The **slk_init** subroutine has the effect of calling the **ripoffline** subroutine to reserve one screen line to accommodate the requested format.

The **slk_set** and **slk_wset** subroutines specify the text of soft label number *labnum*, within the range from 1 to and including 8. The *label* argument is the string to be put on the label. With **slk_set** and **slk_wset**, the width of the label is limited to eight column positions. A null string or a null pointer specifies a blank label. The *justify* argument can have the following values to indicate how to justify label within the space reserved for it:

Item	Description
0	Align the start of label with the start of the space.
1	Center label within the space.
2	Align the end of label with the end of the space.

The **slk_refresh** and **slk_noutrefresh** subroutines correspond to the **wrefresh** and **wnoutrefresh** subroutines.

The **slk_label** subroutine obtains soft label number labnum.

The **slk_clear** subroutine immediately clears the soft labels from the screen.

The **slk_touch** subroutine forces all the soft labels to be output the next time **slk_noutrefresh** or **slk_refresh** subroutines is called.

The **slk_attron, slk_attrset and slk_attroff** subroutines correspond to the **attron**, **attrset**, and **attroff** subroutines. They have an effect only if soft labels are simulated on the bottom line of the screen.

The **slk_attr_off**, **slk_attr_on**, **slk_sttr_set**, and **slk_attroff** subroutines correspond to the **slk_attroff**, **slk_attron**, **slk_attrset**, and **color_set** and thus support the attribute constants with the WA_prefix and color.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as *opts*.

Parameters

Item	Description
attrs	
**opts*	
color_pair_number	
fmt	
labnum	
justify	
**label*	

Examples

For the **slk_init** subroutine:

To initialize soft labels on a terminal that does not support soft labels internally, do the following:

```
slk_init(1);
```

This example arranges the labels so that four labels appear on the right of the screen and four appear on the left.

For the **slk_label** subroutine:

To obtain the label name for soft label 3, use:

```
char *label_name;
label_name = slk_label(3);
```

For the **slk_noutrefresh** subroutine:

To refresh soft label 8 on the virtual screen but not on the physical screen, use:

```
slk_set(8, "Insert", 1);
slk_noutrefresh();
```

For the **slk_refresh** subroutine:

To set and left-justify the soft labels and then refresh the physical screen, use:

```
slk_init(0);
initscr();
slk_set(1, "Insert", 0);
slk_set(2, "Quit", 0);
slk_set(3, "Add", 0);
slk_set(4, "Delete", 0);
slk_set(5, "Undo", 0);
slk_set(6, "Search", 0);
slk_set(7, "Replace", 0);
slk_set(8, "Save", 0);
slk_refresh();
```

For the **slk_set** subroutine:

```
slk_set(2, "Quit", 1);
```

Return Values

Upon successful completion, the **slk_label** subroutine returns the requested label with leading and trailing blanks stripped. Otherwise, it returns a null pointer.

Upon successful completion, the other subroutines return OK. Otherwise, they return ERR.

slk_init Subroutine

Purpose

Initializes soft function-key labels.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

slk_init(*Labfmt***)**
int *Labfmt***;**

Description

The **slk_init** subroutine initializes soft function-key labels. This is one of several subroutines curses provides for manipulating soft function-key labels. These labels appear at the bottom of the screen and give applications, such as editors, a more user-friendly look. To use soft labels, you must call the **slk_init** subroutine before calling the **initscr** or **newterm** subroutine.

Some terminals support soft labels, others do not. For terminals that do not support soft labels. Curses emulates soft labels by using the bottom line of the stdscr. To accommodate soft labels, curses reduces the size of the stdscr and the **LINES** environment variable as required.

Parameter

Item	Description
Labfmt	Simulates soft labels. To arrange three labels on the right, two in the center, and three on the right of the screen, specify a 0 for this parameter. To arrange four labels on the left and four on the right of the screen, specify a 1 for this parameter.

Example

To initialize soft labels on a terminal that does not support soft labels internally, do the following:

```
slk_init(1);
```

This example arranges the labels so that four labels appear on the right of the screen and four appear on the left.

slk_label Subroutine

Purpose

Returns the label name for a specified soft label.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

char *slk_label( LabNum)
int LabNum;
```

Description

The **slk_label** subroutine returns the label name for a specified soft function-key label. These labels appear at the bottom of the screen and give applications, such as editors, a more user-friendly look. The **slk_label** subroutine returns the name in the format it was in when passed to the **slk_set** subroutine. If the name was justified by the **slk_set** subroutine, the justification is removed.

Parameters

Item	Description
LabNum	Specifies the label number. This parameter must be in the range 1 to 8.

Example

To obtain the label name for soft label 3, use:

```
char *label_name;
```

```
label_name = slk_label(3);
```

Return Values

Item	Description
NULL	Indicates a label number that is not valid or a label number not set with the **slk_set** subroutine.
OK	Indicates that the label name was successfully retrieved.

slk_noutrefresh Subroutine

Purpose

Updates the soft labels on the virtual screen.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
slk_noutrefresh()
```

Description

The **slk_noutrefresh** subroutine updates the soft function-key labels on the virtual screen. These labels appear at the bottom of the screen and give applications, such as editors, a more user-friendly look. This subroutine is useful for updating multiple labels. You can use the **slk_noutrefresh** subroutine to update all soft labels on the virtual screen with no updates to the physic al screen. To update the physical screen, use the **slk_refresh** or **refresh** subroutine.

Example

To refresh soft label 8 on the virtual screen but not on the physical screen, use:

```
slk_set(8, "Insert", 1);
slk_noutrefresh();
```

slk_refresh Subroutine

Purpose

Updates soft labels on the virtual and physical screens.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
slk_refresh()
```

Description

The **slk_refresh** subroutine refreshes the virtual and physical screens after an update to soft function-key labels. These labels appear at the bottom of the screen and give applications, such as editors, a more user-friendly look.

Example

To set and left-justify the soft labels and then refresh the physical screen, use:

```
slk_init(0);
initscr();
slk_set(1, "Insert", 0);
slk_set(2, "Quit", 0);
slk_set(3, "Add", 0);
slk_set(4, "Delete", 0);
slk_set(5, "Undo", 0);
slk_set(6, "Search", 0);
slk_set(7, "Replace", 0);
slk_set(8, "Save", 0);
slk_refresh();
```

slk_restore Subroutine

Purpose

Restores soft function-key labels to the screen.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

slk_restore()
```

Description

The **slk_restore** subroutine restores the soft function-key labels to the screen after a call to the **slk_clear** subroutine. The label names are not restored. These labels appear at the bottom of the screen and give applications, such as editors, a more user-friendly look. You must call the **slk_init** subroutine before you can use soft labels.

slk_touch Subroutine

Purpose

Forces an update of the soft function-key labels.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

slk_touch()
```

Description

The **slk_touch** subroutine forces an update of the soft function-key labels on the physical screen the next time the **slk_noutrefresh** subroutine is called. These labels appear at the bottom of the screen and give applications, such as editors, a more user-friendly look. You must call the **slk_init** subroutine before using soft labels.

sockatmark Subroutine

Purpose

Determines whether a socket is at the out-of-band mark.

Syntax

```
#include <sys/socket.h>

 int sockatmark(s)
int s;
```

Description

The **sockatmark** subroutine determines whether the socket specified by the *s* parameter is at the out-of-band data mark. If the protocol for the socket supports out-of-band data by marking the stream with an out-of-band data mark, the **sockatmark** subroutine returns a 1 when all data preceding the mark has been read and the out-of-band data mark is the first element in the receive queue. The **sockatmark** subroutine does not remove the mark from the stream.

The use of this subroutine between receive operations allows an application to determine which received data precedes the out-of-band data and which follows the out-of-band data. There is an inherent race condition in the use of this function. On an empty receive queue, the current read of the location might well be at the mark', but the system has no way of knowing that the next data segment that will arrive from the network will carry the mark, and **sockatmark** will return false The next read operation will silently consume the mark. Because of this, the **sockatmark** subroutine can only be used reliably when the application already knows that the out-of-band data has been seen by the system or that it is known that there is data waiting to be read at the socket.

Parameters

Item	Description
s	Specifies the socked to be checked.

Return Values

Upon successful completion, the **sockatmark** subroutine returns a value indicating whether the socket is at an out-of-band data mark. If the protocol has marked the data stream and all data preceding the mark has been read, the return value is 1. If there is no mark, or if data precedes the mark in the receive

queue, the **sockatmark** subroutine returns a 0. Otherwise, it returns a value of -1 and sets the **errno** global variable to indicate the error.

Error Codes

Item	Description
EBADF	The *s* parameter is not a valid file descriptor.
ENOTTY	The *s* parameter does not specify a descriptor for a socket.

SpmiAddSetHot Subroutine

Purpose

Adds a set of peer statistics values to a hotset.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiHotVals *SpmiAddSetHot(HotSet, StatName,
GrandParent, maxresp,
                                  threshold, frequency, feed_type,
                                  except_type, severity, trap_no)
struct SpmiHotSet *HotSet;
char *StatName;
SpmiCxHdl GrandParent;
int maxresp;
int threshold;
int frequency;
int feed_type;
int excp_type;
int severity;
int trap_no;
```

Description

The **SpmiAddSetHot** subroutine adds a set of peer statistics to a hotset. The **SpmiHotSet** structure that provides the anchor point to the set must exist before the **SpmiAddSetHot** subroutine call can succeed.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

HotSet

Specifies a pointer to a valid structure of type **SpmiHotSet** as created by the **SpmiCreateHotSet** (“SpmiCreateHotSet” on page 1985) subroutine call.

StatName

Specifies the name of the statistic within the subcontexts (peer contexts) of the context identified by the *GrandParent* parameter.

GrandParent

Specifies a valid **SpmiCxHdl** handle as obtained by another subroutine call. The handle must identify a context with at least one subcontext, which contains the statistic identified by the *StatName* parameter. If the context specified is one of the **RTime** contexts, no subcontext need to exist at

the time the **SpmiAddSetHot** subroutine call is issued; the presence of the metric identified by the *StatName* parameter is checked against the context class description.

If the context specified has or may have multiple levels of instantiable context below it (such as the **FS** and **RTime/ARM** contexts), the metric is only searched for at the lowest context level. The **SpmiHotSet** created is a pseudo hotvals structure used to link together a peer group of **SpmiHotVals**structures, which are created under the covers, one for each subcontext of the *GrandParent* context. In the case of **RTime/ARM**, if additional contexts are later added under the *GrandParent* contexts, additional hotsets are added to the peer group. This is transparent to the application program, except that the **SpmiFirstHot**, **SpmiNextHot**, and **SpmiNextHotItem** subroutine calls will return the peer group **SpmiHotVals** pointer rather than the pointer to the pseudo structure.

Note that specifying a specific volume group context (such as **FS/rootvg**) or a specific application context (such as **RTime/ARN/armpeek**) is still valid and won't involve creation of pseudo **SpmiHotVals** structures.

maxresp

Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. If specified as zero, indicates that all **SPMIHotItems** that meet the criteria specified by *threshold* must be returned, up-to a maximum of *maxresp* items. If both exceptions/traps and feeds are requested, the *maxresp* value is used to cap the number of exceptions/alerts as well as the number of items returned. If *feed_type* is specified as **SiHotAlways**, the *maxresp* parameter is still used to return at most *maxresp* items.

Where the *GrandParent* argument specifies a context that has multiple levels of instantiable contexts below it, the *maxresp* is applied to each of the lowest level contexts above the the actual peer contexts at a time. For example, if the *GrandParent* context is **FS** (file systems) and the system has three volume groups, then a *maxresp* value of 2 could cause up to a maximum of 2 x 3 = 6 responses to be generated.

threshold

Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. If specified as zero, indicates that all values read qualify to be returned in feeds. The value specified is compared to the data value read for each peer statistic. If the data value exceeds the *threshold*, it qualifies to be returned as an **SpmiHotItems** element in the **SpmiHotVals** structure. If the *threshold* is specified as a negative value, the value qualifies if it is lower than the numeric value of *threshold*. If *feed_type* is specified as **SiHotAlways**, the threshold value is ignored for feeds. For peer statistics of type **SiCounter**, the *threshold* must be specified as a rate per second; for **SiQuantity** statistics the *threshold* is specified as a level.

frequency

Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. Ignored for feeds. Specifies the minimum number of minutes that must expire between any two exceptions/traps generated from this **SpmiHotVals** structure. This value must be specified as no less than 5 minutes.

feed_type

Specifies if feeds of **SpmiHotItems** should be returned for this **SpmiHotVals** structure. The following values are valid:

SiHotNoFeed
No feeds should be generated

SiHotThreshold
Feeds are controlled by *threshold*.

SiHotAlways
All values, up-to a maximum of *maxresp* must be returned as feeds.

excp_type

Controls the generation of exception data packets and/or the generation of SNMP Traps from **xmservd**. Note that these types of packets and traps can only actually be sent if **xmservd** is running.

Because of this, exception packets and SNMP traps are only generated as long as **xmservd** is active. Traps can only be generated on AIX systems. The conditions for generating exceptions and traps are controlled by the *threshold* and *frequency* parameters. The following values are valid for *excp_type*:

SiNoHotException
Generate neither exceptions not traps.

SiHotException
Generate exceptions but not traps.

SiHotTrap
Generate SNMP traps but not exceptions.

SiHotBoth
Generate both exceptions and SNMP traps.

severity

Required to be positive and greater than zero if exceptions are generated, otherwise specify as zero. Used to assign a severity code to the exception for display by **exmon**.

trap_no

Required to be positive and greater than zero if SNMP traps are generated, otherwise specify as zero. Used to assign the trap number in the generated SNMP trap.

Return Values

The **SpmiAddSetHot** subroutine returns a pointer to a structure of type **SpmiHotVals** if successful. If unsuccessful, the subroutine returns a NULL value.

Programming Notes

The **SpmiAddSetHot** functions in a straight forward manner and as described previously in all cases where the *GrandParent* context is a context that has only one level of instantiable contexts below it. This covers most context types such as CPU, Disk, LAN, etc. In a few cases, currently only the **FS** (file system) and **RTime/ARM** (application response) contexts, the SPMI works by creating pseudo-hotvals structures that effectively expand the hotset. These pseudo-hotvals structures are created either at the time the **SpmiAddSetHot** call is issued or when new subcontexts are created for a context that's already the *GrandParent* of a hotvals peer set. For example:

When a peer set is created for **RTime/ARM**, maybe only a few or no subcontexts of this context exists. If two applications were defined at this point, say **checking** and **savings**, one valsset would be created for the **RTime/ARM** context and a pseudo-valsset for each of **RTime/ARM/checking** and **RTime/ARM/savings**. As new applications are added to the **RTime/ARM** contexts, new pseudo-valssets are automatically added to the hotset.

Pseudo-valssets represent an implementation convenience and also helps minimize the impact of retrieving and presenting data for hotsets. As far as the caller of the **RSiGetHotItem** subroutine call is concerned, it is completely transparent. All this caller will ever see is the real hotvals structure. That is not the case for callers of **SpmiFirstHot**, **SpmiNextHot**, and **SpmiNextHotItem**. All of these subroutines will return pseudo-valssets and the calling program should be prepared to handle this.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined

in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiCreateHotSet

Purpose

Creates an empty hotset.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiHotSet *SpmiCreateHotSet()
```

Description

The **SpmiCreateHotSet** subroutine creates an empty hotset and returns a pointer to an **SpmiHotSet** structure.This structure provides the anchor point for a hotset and must exist before the **SpmiAddSetHot** subroutine can be successfully called.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Return Values

The **SpmiCreateHotSet** subroutine returns a pointer to a structure of type **SpmiHotSet** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiCreateStatSet Subroutine

Purpose

Creates an empty set of statistics.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiStatSet *SpmiCreateStatSet()
```

Description

The **SpmiCreateStatSet** subroutine creates an empty set of statistics and returns a pointer to an **SpmiStatSet** structure.

The **SpmiStatSet** structure provides the anchor point to a set of statistics and must exist before the **SpmiPathAddSetStat** subroutine can be successfully called.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Return Values

The **SpmiCreateStatSet** subroutine returns a pointer to a structure of type **SpmiStatSet** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiDdsAddCx Subroutine

Purpose

Adds a volatile context to the contexts defined by an application.

Library

SPMI Library (**libSpmi**.a)

Syntax

```
#include sys/Spmidef.h
```

```
char *SpmiDdsAddCx(Ix, Path, Descr, Asnno)
ushort Ix;
char *Path, *Descr;
int Asnno;
```

Description

The **SpmiDdsAddCx** subroutine uses the shared memory area to inform the SPMI that a context is available to be added to the context hierarchy, moves a copy of the context to shared memory, and allocates memory for the data area.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

Ix

Specifies the element number of the added context in the table of dynamic contexts. No context can be added if the table of dynamic contexts has not been defined in the **SpmiDdsInit** subroutine call. The first element of the table is element number 0.

Path

Specifies the full path name of the context to be added. If the context is not at the top-level, the parent context must already exist.

Descr

Provides the description of the context to be added as it will be presented to data consumers.

Asnno

Specifies the ASN.1 number to be assigned to the new context. All subcontexts on the same level as the new context must have unique ASN.1 numbers. Typically, each time the **SpmiDdsAddCx** subroutine adds a subcontext to the same parent context, the Asnno parameter is incremented.

Return Values

If successful, the **SpmiDdsAddCx** subroutine returns the address of the shared memory data area. If an error occurs, an error text is placed in the external **SpmiErrmsg** character array, and the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiDdsDelCx Subroutine

Purpose

Deletes a volatile context.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiDdsDelCx(Area)
char *Area;
```

Description

The **SpmiDdsDelCx** subroutine informs the SPMI that a previously added, volatile context should be deleted.

If the SPMI has not detected that the context to delete was previously added dynamically, the **SpmiDdsDelCx** subroutine removes the context from the list of to-be-added contexts and returns the allocated shared memory to the free list. Otherwise, the **SpmiDdsDelCx** subroutine indicates to the SPMI that a context and its associated statistics must be removed from the context hierarchy and any allocated shared memory must be returned to the free list.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

Area

Specifies the address of the previously allocated shared memory data area as returned by an **SpmiDdsAddCx** subroutine call.

Return Values

If successful, the **SpmiDdsDelCx** subroutine returns a value of 0. If an error occurs, an error text is placed in the external **SpmiErrmsg** character array, and the subroutine returns a nonzero value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiDdsInit Subroutine

Purpose

- Establishes a program as a dynamic data-supplier (DDS) program.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
SpmiShare *SpmiDdsInit(CxTab, CxCnt, IxTab, IxCnt,
FileName)
cx_create *CxTab, *IxTab;
int CxCnt, IxCnt;
char *FileName;
```

Description

The SpmiDdsInit subroutine establishes a program as a dynamic data-supplier (DDS) program. To do so, the SpmiDdsInit subroutine:

1. Determines the size of the shared memory required and creates a shared memory segment of that size.

2. Moves all static contexts and all statistics referenced by those contexts to the shared memory.
3. Calls the SPMI and requests it to add all of the DDS static contexts to the context tree.

Note:

1. The **SpmiDdsInit** subroutine issues an **SpmiInit** subroutine call if the application program has not issued one.
2. If the calling program uses shared memory for other purposes, including memory mapping of files, the **SpmiDdsInit** or the **SpmiInit** subroutine call must be issued before access is established to other shared memory areas.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

CxTab

Specifies a pointer to the table of nonvolatile contexts to be added.

CxCnt

Specifies the number of elements in the table of nonvolatile contexts. Use the **CX_L** macro to find this value.

IxTab

Specifies a pointer to the table of volatile contexts the program may want to add later. If no contexts are defined, specify NULL.

IxCnt

Specifies the number of elements in the table of volatile contexts. Use the **CX_L** macro to find this value. If no contexts are defined, specify 0.

FileName

Specifies the fully qualified path and file name to use when creating the shared memory segment. At execution time, if the file exists, the process running the DDS must be able to write to the file. Otherwise, the **SpmiDdsInit** subroutine call does not succeed. If the file does not exist, it is created. If the file cannot be created, the subroutine returns an error. If the file name includes directories that do not exist, the subroutine returns an error.

For non-AIX systems, a sixth argument is required to inform the SPMI how much memory to allocate in the DDS shared memory segment. This is not required for AIX systems because facilities exist to expand a memory allocation in shared memory. The sixth argument is:

size

Size in bytes of the shared memory area to allocate for the DDS program. This parameter is of type int.

Return Values

If successful, the **SpmiDdsInit** subroutine returns the address of the shared memory control area. If an error occurs, an error text is placed in the external **SpmiErrmsg** character array, and the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiDelSetHot Subroutine

Purpose

Removes a single set of peer statistics from a hotset.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiDelSetHot(HotSet, HotVal)
struct SpmiHotSet *HotSet;
struct SpmiHotVals *HotVal;
```

Description

The **SpmiDelSetHot** subroutine removes a single set of peer statistics, identified by the *HotVal* parameter, from a hotset, identified by the *HotSet* parameter.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

HotSet

Specifies a pointer to a valid structure of type **SpmiHotSet**, as created by the “SpmiCreateHotSet” on page 1985 subroutine call.

HotVal

Specifies a pointer to a valid structure of type **SpmiHotVals**, as created by the “SpmiAddSetHot Subroutine” on page 1982 subroutine call. You cannot specify an **SpmiHotVals** that was internally generated by the SPMI library code as described under the *GrandParent* parameter to **SpmiAddSetHot**.

Return Values

The **SpmiDelSetHot** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiDelSetStat Subroutine

Purpose

Removes a single statistic from a set of statistics.

Library

SPMI Library (**libSpmi**.a)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiDelSetStat(StatSet, StatVal)
struct SpmiStatSet *StatSet;
struct SpmiStatVals *StatVal;
```

Description

The **SpmiDelSetStat** subroutine removes a single statistic, identified by the *StatVal* parameter, from a set of statistics, identified by the *StatSet* parameter.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatSet

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the “SpmiCreateStatSet Subroutine” on page 1986 subroutine call.

StatVal

Specifies a pointer to a valid structure of type **SpmiStatVals** as created by the “SpmiPathAddSetStat Subroutine” on page 2016 subroutine call.

Return Values

The **SpmiDelSetStat** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiExit Subroutine

Purpose

Terminates a dynamic data supplier (DDS) or local data consumer program's association with the SPMI, and releases allocated memory.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
void SpmiExit()
```

Description

A successful “SpmiInit Subroutine” on page 2006 or “SpmiDdsInit Subroutine” on page 1989 call allocates shared memory. Therefore, a Dynamic Data Supplier (DDS) program that has issued a successful **SpmiInit** or **SpmiDdsInit** subroutine call should issue an **SpmiExit** subroutine call before the program exits the SPMI. Allocated memory is not released until the program issues an **SpmiExit** subroutine call.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiFirstCx Subroutine

Purpose

Locates the first subcontext of a context.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiCxLink *SpmiFirstCx(CxHandle)
SpmiCxHdl CxHandle;
```

Description

The **SpmiFirstCx** subroutine locates the first subcontext of a context. The subroutine returns a NULL value if no subcontexts are found.

The structure pointed to by the returned pointer contains a handle to access the contents of the corresponding **SpmiCx** structure through the **SpmiGetCx**subroutine call.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

CxHandle

Specifies a valid **SpmiCxHdl** handle as obtained by another subroutine call.

Return Values

The SpmiFirstCx subroutine returns a pointer to an **SpmiCxLink** structure if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiFirstHot Subroutine

Purpose

Locates the first of the sets of peer statistics belonging to a hotset.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiHotVals *SpmiFirstHot(HotSet)
struct SpmiHotSet HotSet;
```

Description

The **SpmiFirstHot** subroutine locates the first of the **SpmiHotVals** structures belonging to the specified **SpmiHotSet**. Using the returned pointer, the **SpmiHotSet** can then either be decoded directly by the calling program, or it can be used to specify the starting point for a subsequent **SpmiNextHotItem** subroutine call. The **SpmiFirstHot** subroutine should only be executed after a successful call to the **SpmiGetHotSet** subroutine.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

HotSet

Specifies a valid **SpmiHotSet** structure as obtained by another subroutine call.

Return Values

The **SpmiFirstHot** subroutine returns a pointer to a structure of type **SpmiHotVals** structure if successful. If unsuccessful, the subroutine returns a NULL value. A returned pointer may refer to a pseudo-hotvals structure as described in the **SpmiAddSetHot** subroutine.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiFirstStat Subroutine

Purpose

Locates the first of the statistics belonging to a context.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiStatLink *SpmiFirstStat(CxHandle)
SpmiCxHdl CxHandle;
```

Description

The **SpmiFirstStat** subroutine locates the first of the statistics belonging to a context. The subroutine returns a NULL value if no statistics are found.

The structure pointed to by the returned pointer contains a handle to access the contents of the corresponding **SpmiStat** structure through the "SpmiGetStat Subroutine" on page 2002 call.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

CxHandle

Specifies a valid **SpmiCxHdl** handle as obtained by another subroutine call.

Return Values

The **SpmiFirstStat** subroutine returns a pointer to a structure of type **SpmiStatLink** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiFirstVals Subroutine

Purpose

Returns a pointer to the first **SpmiStatVals** structure belonging to a set of statistics.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiStatVals *SpmiFirstVals(StatSet)
struct SpmiStatSet *StatSet;
```

Description

The **SpmiFirstVals** subroutine returns a pointer to the first **SpmiStatVals** structure belonging to the set of statistics identified by the *StatSet* parameter. **SpmiStatVals** structures are accessed in reverse order so the last statistic added to the set of statistics is the first one returned. This subroutine call should only be issued after an **SpmiGetStatSet** subroutine has been issued against the statset.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatSet

Specifies a pointer to a valid structure of type **Spmi**StatSet as created by the **Spmi**CreateStatSet subroutine call.

Return Values

The **SpmiFirstVals** subroutine returns a pointer to an **SpmiStatVals** structure if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiFreeHotSet Subroutine

Purpose

Erases a hotset.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiFreeHotSet(HotSet)
struct SpmiHotSet *HotSet;
```

Description

The **SpmiFreeHotSet** subroutine erases the hotset identified by the *HotSet* parameter. All **SpmiHotVals** structures chained off the **SpmiHotSet** structure are deleted before the set itself is deleted.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

HotSet

Specifies a pointer to a valid structure of type **SpmiHotSet** as created by the “SpmiCreateHotSet” on page 1985 subroutine call.

Return Values

The **SpmiFreeHotSet** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiFreeStatSet Subroutine

Purpose

Erases a set of statistics.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiFreeStatSet(StatSet)
struct SpmiStatSet *StatSet;
```

Description

The **SpmiFreeStatSet** subroutine erases the set of statistics identified by the *StatSet* parameter. All **SpmiStatVals** structures chained off the **SpmiStatSet** structure are deleted before the set itself is deleted.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatSet

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the **SpmiCreateStatSet** subroutine call.

Return Values

The **SpmiFreeStatSet** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiGetCx Subroutine

Purpose

Returns a pointer to the **SpmiCx** structure corresponding to a specified context handle.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiCx *SpmiGetCx(CxHandle)
SpmiCxHdl CxHandle;
```

Description

The **SpmiGetCx** subroutine returns a pointer to the **SpmiCx** structure corresponding to the context handle identified by the *CxHandle* parameter.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

CxHandle

Specifies a valid **SpmiCxHdl** handle as obtained by another subroutine call.

Return Values

The **SpmiGetCx** subroutine returns a a pointer to an **SpmiCx** data structure if successful. If unsuccessful, the subroutine returns NULL.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiGetHotSet Subroutine

Purpose

Requests the SPMI to read the data values for all sets of peer statistics belonging to a specified **SpmiHotSet**.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiGetHotSet(HotSet, Force);
struct SpmiHotSet *HotSet;
boolean Force;
```

Description

The **SpmiGetHotSet** subroutine requests the SPMI to read the data values for all peer sets of statistics belonging to the **SpmiHotSet** identified by the *HotSet* parameter. The *Force* parameter is used to force the data values to be refreshed from their source.

The *Force* parameter works by resetting a switch held internally in the SPMI for all **SpmiStatVals** and **SpmiHotVals** structures, regardless of the **SpmiStatSet**s and **SpmiHotSet**s to which they belong. Whenever the data value for a peer statistic is requested, this switch is checked. If the switch is set, the SPMI reads the latest data value from the original data source. If the switch is not set, the SPMI reads the data value stored in the **SpmiHotVals** structure. This mechanism allows a program to synchronize and minimize the number of times values are retrieved from the source. One method programs can use is to ensure the force request is not issued more than once per elapsed amount of time.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

HotSet

Specifies a pointer to a valid structure of type **SpmiHotSet** as created by the “SpmiCreateHotSet” on page 1985 subroutine call.

Force

If set to true, forces a refresh from the original source before the SPMI reads the data values for the set. If set to false, causes the SPMI to read the data values as they were previously retrieved from the data source.

When the force argument is set true, the effect is that of marking all statistics known by the SPMI as obsolete, which causes the SPMI to refresh all requested statistics from kernel memory or other sources. As each statistic is refreshed, the obsolete mark is reset. Statistics that are not part of the **HotSet** specified in the subroutine call remain marked as obsolete. Therefore, if an application

repetitively issues a series of, **SpmiGetHotSet** and **SpmiGetStatSet** subroutine calls for multiple hotsets and statsets, each time, only the first such call need set the force argument to true.

Return Values

The **SpmiGetHotSet** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiGetStat Subroutine

Purpose

Returns a pointer to the **SpmiStat** structure corresponding to a specified statistic handle.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiStat *SpmiGetStat(StatHandle)
SpmiStatHdl StatHandle;
```

Description

The **SpmiGetStat** subroutine returns a pointer to the **SpmiStat** structure corresponding to the statistic handle identified by the *StatHandle* parameter.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatHandle

Specifies a valid **SpmiStatHdl** handle as obtained by another subroutine call.

Return Values

The **SpmiGetStat** subroutine returns a pointer to a structure of type **SpmiStat** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiGetStatSet Subroutine

Purpose

Requests the SPMI to read the data values for all statistics belonging to a specified set.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiGetStatSet(StatSet, Force);
struct SpmiStatSet *StatSet;
boolean Force;
```

Description

The **SpmiGetStatSet** subroutine requests the SPMI to read the data values for all statistics belonging to the **SpmiStatSet** identified by the *StatSet* parameter. The *Force* parameter is used to force the data values to be refreshed from their source.

The *Force* parameter works by resetting a switch held internally in the SPMI for all **SpmiStatVals** and **SpmiHotVals** structures, regardless of the **SpmiStatSets** and **SpmiHotSets** to which they belong. Whenever the data value for a statistic is requested, this switch is checked. If the switch is set, the SPMI reads the latest data value from the original data source. If the switch is not set, the SPMI reads the data value stored for the **SpmiStatVals** structure. This mechanism allows a program to synchronize and minimize the number of times values are retrieved from the source. One method is to ensure the force request is not issued more than once per elapsed amount of time.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatSet

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the **SpmiCreateStatSet** subroutine call.

Force

If set to true, forces a refresh from the original source before the SPMI reads the data values for the set. If set to false, causes the SPMI to read the data values as they were previously retrieved from the data source.

When the force argument is set true, the effect is that of marking all statistics known by the SPMI as obsolete, which causes the SPMI to refresh all requested statistics from kernel memory or other sources. As each statistic is refreshed, the obsolete mark is reset. Statistics that are not part of the **StatSet** specified in the subroutine call remain marked as obsolete. Therefore, if an application repetitively issues the **SpmiGetStatSet** and **SpmiGetHotSet** subroutine calls for multiple statsets and hotsets, each time, only the first such call need set the force argument to true.

Return Values

The **Spmi**GetStatSet subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiGetValue Subroutine

Purpose

Returns a decoded value based on the type of data value extracted from the data field of an **SpmiStatVals** structure.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
float SpmiGetValue(StatSet, StatVal)
struct SpmiStatSet *StatSet;
struct SpmiStatVals *StatVal;
```

Description

The **SpmiGetValue** subroutine performs the following steps:

1. Verifies that an **SpmiStatVals** structure exists in the set of statistics identified by the *StatSet* parameter.
2. Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing.
3. Determines the data value as being of either type **SiQuantity** or type **SiCounter**.
4. If the data value is of type **SiQuantity**, returns the **val** field of the **SpmiStatVals** structure.
5. If the data value is of type **SiCounter**, returns the value of the **val_change** field of the **SpmiStatVals** structure divided by the elapsed number of seconds since the previous time a data value was requested for this set of statistics.

This subroutine call should only be issued after an **SpmiGetStatSet** subroutine has been issued against the statset.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatSet

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the **SpmiCreateStatSet** subroutine call.

StatVal

Specifies a pointer to a valid structure of type **SpmiStatVals** as created by the **SpmiPathAddSetStat** subroutine call or returned by the **SpmiFirstVals** or **SpmiNextVals** subroutine calls.

Return Values

The **SpmiGetValue** subroutine returns the decoded value if successful. If unsuccessful, the subroutine returns a negative value that has a numerical value of at least 1.1.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiInit Subroutine

Purpose

Initializes the SPMI for a local data consumer program.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiInit (TimeOut)
int TimeOut;
```

Description

The **SpmiInit** subroutine initializes the SPMI. During SPMI initialization, a memory segment is allocated and the application program obtains basic addressability to that segment. An application program must issue the **SpmiInit** subroutine call before issuing any other subroutine calls to the SPMI.

Note: The **SpmiInit** subroutine is automatically issued by the **SpmiDdsInit** subroutine call. Successive **SpmiInit** subroutine calls are ignored.

Note: If the calling program uses shared memory for other purposes, including memory mapping of files, the **SpmiInit** subroutine call must be issued before access is established to other shared memory areas.

The SPMI entry point called by the **SpmiInit** subroutine assigns a segment register to be used by the SPMI subroutines (and the application program) for accessing common shared memory and establishes the access mode to the common shared memory segment. After SPMI initialization, the SPMI subroutines are able to access the common shared memory segment in read-only mode.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

TimeOut

Specifies the number of seconds the SPMI waits for a Dynamic Data Supplier (DDS) program to update its shared memory segment. If a DDS program does not update its shared memory segment in the time specified, the SPMI assumes that the DDS program has terminated or disconnected from shared memory and removes all contexts and statistics added by the DDS program.

The SPMI saves the largest *TimeOut* value received from the programs that invoke the SPMI. The *TimeOut* value must be zero or must be greater than or equal to 15 seconds and less than or equal to 600 seconds. A value of zero overrides any other value from any other program that invokes the SPMI and disables the checking for terminated DDS programs.

Return Values

The **SpmiInit** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value. If a nonzero value is returned, the application program should not attempt to issue additional SPMI subroutine calls.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiInstantiate Subroutine

Purpose

Explicitly instantiates the subcontexts of an instantiable context.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
int SpmiInstantiate(CxHandle)
SpmiCxHdl CxHandle;
```

Description

The **SpmiInstantiate** subroutine explicitly instantiates the subcontexts of an instantiable context. If the context is not instantiable, do not call the **SpmiInstantiate** subroutine.

An instantiation is done implicitly by the **SpmiPathGetCx** and **SpmiFirstCx** subroutine calls. Therefore, application programs usually do not need to instantiate explicitly.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

CxHandle

Specifies a valid context handle **SpmiCxHdl** as obtained by another subroutine call.

Return Values

The **SpmiInstantiate** subroutine returns a value of 0 if successful. If the context is not instantiable, the subroutine returns a nonzero value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiNextCx Subroutine

Purpose

Locates the next subcontext of a context.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiCxLink *SpmiNextCx(CxLink )struct SpmiCxLink *CxLink;
```

Description

The **SpmiNextCx** subroutine locates the next subcontext of a context, taking the context identified by the *CxLink* parameter as the current subcontext. The subroutine returns a NULL value if no further subcontexts are found.

The structure pointed to by the returned pointer contains an **SpmiCxHdl** handle to access the contents of the corresponding **SpmiCx** structure through the **SpmiGetCx** subroutine call.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

CxLink

Specifies a pointer to a valid **SpmiCxLink** structure as obtained by a previous **SpmiFirstCx** subroutine.

Return Values

The **SpmiNextCx** subroutine returns a pointer to a structure of type **SpmiCxLink** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiNextHot Subroutine

Purpose

Locates the next set of peer statistics **SpmiHotVals** belonging to an **SpmiHotSet**.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiHotVals *SpmiNextHot(HotSet, HotVals)
struct SpmiHotSet *HotSet;
struct SpmiHotVals *HotVals;
```

Description

The **SpmiNextHot** subroutine locates the next **SpmiHotVals** structure belonging to an **SpmiHotSet**, taking the set of peer statistics identified by the *HotVals* parameter as the current one. The subroutine returns a NULL value if no further **SpmiHotVals** structures are found. The **SpmiNextHot** subroutine should only be executed after a successful call to the **SpmiGetHotSet** subroutine and (usually, but not necessarily) a call to the **SpmiFirstHot** subroutine and one or more subsequent calls to **SpmiNextHot**.

The subroutine allows the application programmer to position at the next set of peer statistics in preparation for using the **SpmiNextHotItem** subroutine call to traverse this peer set's array of **SpmiHotItems** elements. Use of this subroutine is only necessary if it is desired to skip over some **SpmiHotVals** structures in an **SpmiHotSet**. Under most circumstances, the **SpmiNextHotItem** will be the sole means of accessing all elements of the **SpmiHotItems** arrays of all peer sets belonging to an **SpmiHotSet**.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

HotSet

Specifies a valid pointer to an **SpmiHotSet** structure as obtained by a previous "SpmiCreateHotSet" on page 1985 call.

HotVals

Specifies a pointer to an **SpmiHotVals** structure as returned by a previous **SpmiFirstHot** or **SpmiNextHot** subroutine call or as returned by an **SpmiAddSetHot** subroutine call.

Return Values

The **SpmiNextHot** subroutine returns a pointer to the next **SpmiHotVals** structure within the hotset. If no more **SpmiHotVals** structures are available, the subroutine returns a NULL value. A returned pointer may refer to a pseudo-hotvals structure as described the **SpmiAddSetHot** subroutine.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiNextHotItem Subroutine

Purpose

Locates and decodes the next **SpmiHotItems** element at the current position in an **SpmiHotSet**.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiHotVals *SpmiNextHotItem(HotSet, HotVals, index,
value, name)
struct SpmiHotSet *HotSet;
struct SpmiHotVals *HotVals;
int *index;
```

```
float *value;
char **name;
```

Description

The **SpmiNextHotItem** subroutine locates the next **SpmiHotItems** structure belonging to an **SpmiHotSet**, taking the element identified by the *HotVals* and *index* parameters as the current one. The subroutine returns a NULL value if no further **SpmiHotItems** structures are found. The **SpmiNextHotItem** subroutine should only be executed after a successful call to the **SpmiGetHotSet** subroutine.

The **SpmiNextHotItem** subroutine is designed to be used for walking all **SpmiHotItems** elements returned by a call to the **SpmiGetHotSet** subroutine, visiting the **SpmiHotVals** structures one by one. By feeding the returned value and the updated integer pointed to by *index* back to the next call, this can be done in a tight loop. Successful calls to **SpmiNextHotItem** will decode each **SpmiHotItems** element and return the data value in *value* and the name of the peer context that owns the corresponding statistic in *name*.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

HotSet

Specifies a valid pointer to an **SpmiHotSet** structure as obtained by a previous "SpmiCreateHotSet" on page 1985 call.

HotVals

Specifies a pointer to an **SpmiHotVals** structure as returned by a previous**SpmiNextHotItem**, **SpmiFirstHot**, or **SpmiNextHot** subroutine call or as returned by an **SpmiAddSetHot** subroutine call. If this parameter is specified as NULL, the first **SpmiHotVals** structure of the **SpmiHotSet** is used and the *index* parameter is assumed to be set to zero, regardless of its actual value.

index

A pointer to an integer that contains the desired element number in the **SpmiHotItems** array of the **SpmiHotVals** structure specified by *HotVals*. A value of zero points to the first element. When the **SpmiNextHotItem** subroutine returns, the integer contain the index of the next **SpmiHotItems** element within the returned **SpmiHotVals** structure. If the last element of the array is decoded, the value in the integer will point beyond the end of the array, and the **SpmiHotVals** pointer returned will point to the peer set, which has now been completely decoded. By passing the returned **SpmiHotVals** pointer and the *index* parameter to the next call to **SpmiNextHotItem**, the subroutine will detect this and proceed to the first **SpmiHotItems** element of the next **SpmiHotVals** structure if one exists.

value

A pointer to a float variable. A successful call will return the decoded data value for the statistic. Before the value is returned, the **SpmiNextHotItem** function:

- Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing.
- Determines the data value as being either type **SiQuantity** or type **SiCounter** and performs one of the actions listed here:
  - If the data value is of type **SiQuantity**, the subroutine returns the **val** field of the **SpmiHotItems** structure.
  - If the data value is of type **SiCounter**, the subroutine returns the value of the **val_change** field of the **SpmiHotItems** structure divided by the elapsed number of seconds since the previous time a data value was requested for this set of statistics.

name

A pointer to a character pointer. A successful call will return a pointer to the name of the peer context for which the data value was read.

Return Values

The **SpmiNextHotItem** subroutine returns a pointer to the current **SpmiHotVals** structure within the hotset. If no more **SpmiHotVals** structures are available, the subroutine returns a NULL value. The structure returned contains the data, such as threshold, which may be relevant for presentation of the results of an **SpmiGetHotSet** subroutine call to end-users. A returned pointer may refer to a pseudo-hotvals structure as described in the **SpmiAddSetHot** subroutine.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiNextStat Subroutine

Purpose

Locates the next statistic belonging to a context.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiStatLink *SpmiNextStat(StatLink)
struct SpmiStatLink *StatLink;
```

Description

The **SpmiNextStat** subroutine locates the next statistic belonging to a context, taking the statistic identified by the *StatLink* parameter as the current statistic. The subroutine returns a NULL value if no further statistics are found.

The structure pointed to by the returned pointer contains an **SpmiStatHdl** handle to access the contents of the corresponding **SpmiStat** structure through the “SpmiGetStat Subroutine” on page 2002 call.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatLink

Specifies a valid pointer to a **SpmiStatLink** structure as obtained by a previous "SpmiFirstStat Subroutine" on page 1996 call.

Return Values

The **SpmiNextStat** subroutine returns a pointer to a structure of type **SpmiStatLink** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiNextVals Subroutine

Purpose

Returns a pointer to the next **SpmiStatVals** structure in a set of statistics.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiStatVals *SpmiNextVals(StatSet, StatVal)
struct SpmiStatSet *StatSet;
struct SpmiStatVals *StatVal;
```

Description

The **SpmiNextVals** subroutine returns a pointer to the next **SpmiStatVals** structure in a set of statistics, taking the structure identified by the *StatVal* parameter as the current structure. The **SpmiStatVals** structures are accessed in reverse order so the statistic added before the current one is returned. This subroutine call should only be issued after an **SpmiGetStatSet** subroutine has been issued against the statset.

Parameters

StatSet

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the "SpmiCreateStatSet Subroutine" on page 1986 call.

StatVal

Specifies a pointer to a valid structure of type **SpmiStatVals** as created by the "SpmiPathAddSetStat Subroutine" on page 2016 subroutine call or returned by a previous "SpmiFirstVals Subroutine" on page 1997 or **SpmiNextVals** subroutine call.

Return Values

The **SpmiNextVals** subroutine returns a pointer to a **SpmiStatVals** structure if successful. If unsuccessful, the subroutine returns a NULL value.

SpmiNextValue Subroutine

Purpose

Returns either the first **SpmiStatVals** structure in a set of statistics or the next **SpmiStatVals** structure in a set of statistics and a decoded value based on the type of data value extracted from the data field of an **SpmiStatVals** structure.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiStatVals*SpmiNextValue( StatSet, StatVal, value)
struct SpmiStatSet *StatSet;
struct SpmiStatVals *StatVal;
float *value;
```

Description

Instead of issuing subroutine calls to "SpmiFirstVals Subroutine" on page 1997 / "SpmiNextVals Subroutine" on page 2013 (to get the first or next **SpmiStatVals** structure) followed by calls to **SpmiGetValue** (to get the decoded value from the **SpmiStatVals** structure), the **SpmiNextValue** subroutine returns both in one call. This subroutine call returns a pointer to the first **SpmiStatVals** structure belonging to the *StatSet* parameter if the *StatVal* parameter is NULL. If the *StatVal* parameter is not NULL, the next **SpmiStatVals** structure is returned, taking the structure identified by the *StatVal* parameter as the current structure. The data value corresponding to the returned **SpmiStatVals** structure is decoded and returned in the field pointed to by the value argument. In decoding the data value, the subroutine does the following:

- Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing.
- Determines the data value as being either type **SiQuantity** or type **SiCounter** and performs one of the actions listed here:
  - If the data value is of type **SiQuantity**, the subroutine returns the **val** field of the **SpmiStatVals** structure.

– If the data value is of type **SiCounter**, the subroutine returns the value of the **val_change** field of the **SpmiStatVals** structure divided by the elapsed number of seconds since the previous time a data value was requested for this set of statistics.

Note: This subroutine call should only be issued after an "SpmiGetStatSet Subroutine" on page 2003 has been issued against the statset.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatSet

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the "SpmiCreateStatSet Subroutine" on page 1986 call.

StatVal

Specifies either a NULL pointer or a pointer to a valid structure of type **SpmiStatVals** as created by the "SpmiPathAddSetStat Subroutine" on page 2016 call or returned by a previous **SpmiNextValue** subroutine call. If *StatVal* is NULL, then the first **SpmiStatVals** pointer belonging to the set of statistics pointed to by *StatSet* is returned.

valueA pointer used to return a decoded value based on the type of data value extracted from the data field of the returned **SpmiStatVals** structure.

Return Value

The **SpmiNextValue** subroutine returns a pointer to a **SpmiStatVals** structure if successful. If unsuccessful, the subroutine returns a NULL value.

If the **StatVal** parameter is:

NULL The first **SpmiStatVals** structure belonging to the **StatSet** parameter is returned.

not NULL The next **SpmiStatVals** structure after the structure identified by the **StatVal** parameter is returned and the value parameter is used to return a decoded value based on the type of data value extracted from the data field of the returned **SpmiStatVals** structure.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Programming Notes

The **SpmiNextValue** subroutine maintains internal state information so that retrieval of the next data value from a statset can be done without traversing linked lists of data structures. The stats information is kept separate for each process, but is shared by all threads of a process.

If the subroutine is accessed from multiple threads, the state information is useless and the performance advantage is lost. The same is true if the program is simultaneously accessing two or more statsets. To benefit from the performance advantage of the **SpmiNextValue** subroutine, a program should retrieve all values in order from one stat set before retrieving values from the next statset.

The implementation of the subroutine allows a program to retrieve data values beginning at any point in the statset if the **SpmiStatVals** pointer is known. Doing so will cause a linked list traversal. If subsequent invocations of **SpmiNextValue** uses the value returned from the first and following invocation as their second argument, the traversal of the link list can be avoided.

It should be noted that the value returned by a successful **SpmiNextValue** invocation is always the pointer to the **SpmiStatVals** structure whose data value is decoded and returned in the value argument.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiPathAddSetStat Subroutine

Purpose

Adds a statistics value to a set of statistics.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
struct SpmiStatVals *SpmiPathAddSetStat(StatSet, StatName,
Parent)
struct SpmiStatSet *StatSet;
char *StatName;
SpmiCxHdl Parent;
```

Description

The **SpmiPathAddSetStat** subroutine adds a statistics value to a set of statistics. The **SpmiStatSet** structure that provides the anchor point to the set must exist before the **SpmiPathAddSetStat** subroutine call can succeed.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

StatSet

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the "SpmiCreateStatSet Subroutine" on page 1986 call.

StatName

Specifies the name of the statistic within the context identified by the *Parent* parameter.If the *Parent* parameter is NULL, you must specify the fully qualified path name of the statistic in the *StatName* parameter.

Parent

Specifies either a valid **SpmiCxHdl** handle as obtained by another subroutine call or a NULL value.

Return Values

The **SpmiPathAddSetStat** subroutine returns a pointer to a structure of type **SpmiStatVals** if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiPathGetCx Subroutine

Purpose

Returns a handle to use when referencing a context.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h
```

```
SpmiCxHdl SpmiPathGetCx(CxPath, Parent)
char *CxPath;
SpmiCxHdl Parent;
```

Description

The **SpmiPathGetCx** subroutine searches the context hierarchy for a given path name of a context and returns a handle to use when subsequently referencing the context.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

CxPath

Specifies the path name of the context to find. If you specify the fully qualified path name in the *CxPath* parameter, you must set the *Parent* parameter to NULL. If the path name is not qualified or is only partly qualified (that is, if it does not include the names of all contexts higher in the data hierarchy), the **SpmiPathGetCx** subroutine begins searching the hierarchy at the context identified

by the *Parent* parameter. If the *CxPath* parameter is either NULL or an empty string, the subroutine returns a handle identifying the Top context.

Parent

Specifies the anchor context that fully qualifies the *CxPath* parameter. If you specify a fully qualified path name in the *CxPath* parameter, you must set the *Parent* parameter to NULL.

Return Values

The **SpmiPathGetCx** subroutine returns a handle to a context if successful. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

SpmiStatGetPath Subroutine

Purpose

Returns the full path name of a statistic.

Library

SPMI Library (**libSpmi.a**)

Syntax

```
#include sys/Spmidef.h>
```

```
char *miStatGetPath(Parent, StatHandle, MaxLevels)
SpmiCxHdlSp Parent;
SpmiStatHdl StatHandle;
int  MaxLevels;
```

Description

The **SpmiStatGetPath** subroutine returns the full path name of a statistic, given a parent context **SpmiCxHdl** handle and a statistics **SpmiStatHdl** handle. The *MaxLevels* parameter can limit the number of levels in the hierarchy that must be searched to generate the path name of the statistic.

The memory area pointed to by the returned pointer is freed when the **SpmiStatGetPath** subroutine call is repeated. For each invocation of the subroutine, a new memory area is allocated and its address returned.If the calling program needs the returned character string after issuing the **SpmiStatGetPath** subroutine call, the program must copy the returned string to locally allocated memory before reissuing the subroutine call.

This subroutine is part of the server option of the Performance Aide for AIX licensed product.

Parameters

Parent

Specifies a valid **SpmiCxHdl** handle as obtained by another subroutine call.

StatHandle

Specifies a valid **SpmiStatHdl** handle as obtained by another subroutine call. This handle must point to a statistic belonging to the context identified by the *Parent* parameter.

MaxLevels

Limits the number of levels in the hierarchy that must be searched to generate the path name. If this parameter is set to 0, no limit is imposed.

Return Values

If successful, the **SpmiStatGetPath** subroutine returns a pointer to a character array containing the full path name of the statistic. If unsuccessful, the subroutine returns a NULL value.

Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error.

Files

Item	Description
/usr/include/sys/Spmidef.h	Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

sqrt, sqrtf, sqrtl, sqrtd32, sqrtd64, and sqrtd128 Subroutines

Purpose

Computes the square root.

Syntax

```
#include <math.h>
double sqrt ( x)
double x;
```

```
float sqrtf (x)
float x;
```

```
long double sqrtl (x)
long double x;
```

```
_Decimal32 sqrtd32 (x)
_Decimal32 x;

_Decimal64 sqrtd64 (x)
_Decimal64 x;

_Decimal128 sqrtd128 (x)
_Decimal128 x;
```

Description

The **sqrt**, **sqrtf**, **sqrtl**, **sqrtd32**, **sqrtd64**, and **sqrtd128** subroutines compute the square root of the *x* parameter.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies some double-precision floating-point value.

Return Values

Upon successful completion, the **sqrt**, **sqrtf**, **sqrtl**, **sqrtd32**, **sqrtd64**, and **sqrtd128** subroutines return the square root of *x*.

For finite values of *x* < -0, a domain error occurs, and a NaN is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or +Inf, *x* is returned.

If *x* is -Inf, a domain error shall occur, and a NaN is returned.

Error Codes

When using **libm.a** (**-lm**):

For the **sqrt** subroutine, if the value of *x* is negative, a NaNQ is returned and the **errno** global variable is set to a **EDOM** value.

When using **libmsaa.a** (**-lmsaa**):

If the value of *x* is negative, a 0 is returned and the **errno** global variable is set to a **EDOM** value. A message indicating a **DOMAIN** error is printed on the standard error output.

These error-handling procedures may be changed with the **matherr** subroutine when using the **libmsaa.a** (**-lmsaa**) library.

src_err_msg Subroutine

Purpose

Retrieves a System Resource Controller (SRC) error message.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
int src_err_msg ( errno,  ErrorText)
int errno;
char **ErrorText;
```

Description

The **src_err_msg** subroutine retrieves a System Resource Controller (SRC) error message.

Parameters

Item	Description
errno	Specifies the SRC error code.
ErrorText	Points to a character pointer to place the SRC error message.

Return Values

Upon successful completion, the **src_err_msg** subroutine returns a value of 0. Otherwise, a value of -1 is returned. No error message is returned.

src_err_msg_r Subroutine

Purpose

Gets the System Resource Controller (SRC) error message corresponding to the specified SRC error code.

Library

System Resource Controller (**libsrc.a**)

Syntax

```
#include <spc.h>

int src_err_msg_r (srcerrno, ErrorText)
int  srcerrno;
char ** ErrorText;
```

Description

The **src_err_msg_r** subroutine returns the message corresponding to the input srcerrno value in a caller-supplied buffer. This subroutine is threadsafe and reentrant.

Parameters

Item	Description
srcerrno	Specifies the SRC error code.
ErrorText	Pointer to a variable containing the address of a caller-supplied buffer where the message will be returned. If the length of the message is unknown, the maximum message length can be used when allocating the buffer. The maximum message length is **SRC_BUF_MAX** in **/usr/include/spc.h** (2048 bytes).

Return Values

Upon successful completion, the **src_err_msg_r** subroutine returns a value of 0. Otherwise, no error message is returned and the subroutine returns a value of -1.

srcrrqs Subroutine

Purpose

Gets subsystem reply information from the System Resource Controller (SRC) request received.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <spc.h>

struct srchdr *srcrrqs ( Packet)
char *Packet;
```

Description

The **srcrrqs** subroutine saves the **srchdr** information contained in the packet the subsystem received from the System Resource Controller (SRC). The **srchdr** structure is defined in the **spc.h** file. This routine must be called by the subsystem to complete the reception process of any packet received from the SRC. The subsystem requires this information to reply to any request that the subsystem receives from the SRC.

Note: The saved **srchdr** information is overwritten each time this subroutine is called.

Parameters

Item	Description
Packet	Points to the SRC request packet received by the subsystem. If the subsystem received the packet on a message queue, the *Packet* parameter must point past the message type of the packet to the start of the request information. If the subsystem received the information on a socket, the *Packet* parameter points to the start of the packet received on the socket.

Return Values

The **srcrrqs** subroutine returns a pointer to the static **srchdr** structure, which contains the return address for the subsystem response.

Examples

The following will obtain the subsystem reply information:

```
int rc;
struct sockaddr addr;
int addrsz;
struct srcreq packet;

/* wait to receive packet from SRC daemon */
rc=recvfrom(0, &packet, sizeof(packet), 0, &addr, &addrsz);
/* grab the reply information from the SRC packet */
if (rc>0)
   srchdr=srcrrqs (&packet);
```

Files

Item	Description
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

srcrrqs_r Subroutine

Purpose

Copies the System Resource Controller (SRC) request header to the specified buffer. The SRC request header contains the return address where the caller sends responses for this request.

Library

System Resource Controller (**libsrc.a**)

Syntax

```
#include <spc.h>

struct srchdr *srcrrqs_r (Packet, SRChdr)
char * Packet;
struct srchdr * SRChdr;
```

Description

The **srcrrqs_r** subroutine saves the SRC request header (srchdr) information contained in the packet the subsystem received from the Source Resource Controller. The **srchdr** structure is defined in the **spc.h** file. This routine must be called by the subsystem to complete the reception process of any packet received from the SRC. The subsystem requires this information to reply to any request that the subsystem receives from the SRC.

This subroutine is threadsafe and reentrant.

Parameters

Item	Description
Packet	Points to the SRC request packet received by the subsystem. If the subsystem received the packet on a message queue, the *Packet* parameter must point past the message type of the packet to the start of the request information. If the subsystem received the information on a socket, the *Packet* parameter points to the start of the packet received on the socket.
SRChdr	Points to a caller-supplied buffer. The **srcrrqs_r** subroutine copies the request header to this buffer.

Examples

The following will obtain the subsystem reply information:

```
int rc;
struct sockaddr addr;
int addrsz;
struct srcreq packet;
struct srchdr *header;
struct srchdr *rtn_addr;

/*wait to receive packet from SRC daemon */
rc=recvfrom(0, &packet, sizeof(packet), 0, &addr, &addrsz;
/* grab the reply information from the SRC packet */
if (rc>0)
{
  header = (struct srchdr *)malloc(sizeof(struct srchdr));
  rtn_addr = srcrrqs_r(&packet,header);
  if (rtn_addr == NULL)
 {
    /* handle error */
    .
    .
  }
```

Return Values

Upon successful completion, the **srcrrq_r** subroutine returns the address of the caller-supplied buffer.

Error Codes

If either of the input addresses is NULL, the **srcrrqs_r** subroutine fails and returns a value of NULL.

Item	Description
SRC_PARM	One of the input addresses is NULL.

srcsbuf Subroutine

Purpose

Gets status for a subserver or a subsystem and returns status text to be printed.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

#include <spc.h>

intsrcsbuf(*Host*,*Type*,*SubsystemName*,

```
SubserverObject,SubsystemPID, StatusType,StatusFrom,StatusText,Continued)

char * Host, * SubsystemName;

char * SubserverObject, ** StatusText;

short  Type,  StatusType;
int  SubsystemPID,  StatusFrom, * Continued;
```

Description

The **srcsbuf** subroutine gets the status of a subserver or subsystem and returns printable text for the status in the address pointed to by the *StatusText* parameter.

When the *StatusType* parameter is **SHORTSTAT** and the *Type* parameter is **SUBSYSTEM**, the **srcstat** subroutine is called to get the status of one or more subsystems. When the *StatusType* parameter is **LONGSTAT** and the *Type* parameter is **SUBSYSTEM**, the **srcrsqt** subroutine is called to get the long status of one subsystem. When the *Type* parameter is not **SUBSYSTEM**, the **srcsrqt** subroutine is called to get the long or short status of a subserver.

Parameters

Item	Description
Host	Specifies the foreign host on which this status action is requested. If the host is null, the status request is sent to the System Resource Controller (SRC) on the local host. The local user must be running as "root". The remote system must be configured to accept remote System Resource Controller requests. That is, the **srcmstr** daemon (see **/etc/inittab**) must be started with the **-r** flag and the **/etc/hosts.equiv** or **.rhosts** file must be configured to allow remote requests.
Type	Specifies whether the status request applies to the subsystem or subserver. If the *Type* parameter is set to **SUBSYSTEM**, the status request is for a subsystem. If not, the status request is for a subserver and the *Type* parameter is a subserver code point.
SubsystemName	Specifies the name of the subsystem on which to get status. To get the status of all subsystems, use the **SRCALLSUBSYS** constant. To get the status of a group of subsystems, the *SubsystemName* parameter must start with the **SRCGROUP** constant, followed by the name of the group for which you want status appended. If you specify a null *SubsystemName* parameter, you must specify a *SubsystemPID* parameter.
SubserverObject	Specifies a subserver object. The *SubserverObject* parameter modifies the *Type* parameter. The *SubserverObject* parameter is ignored if the *Type* parameter is set to **SUBSYSTEM**. The use of the *SubserverObject* parameter is determined by the subsystem and the caller. This parameter will be placed in the `objname` field of the **subreq** structure that is passed to the subsystem.
SubsystemPID	Specifies the process ID of the subsystem on which to get status, as returned by the **srcstrt** subroutine. You must specify the *SubsystemPID* parameter if multiple instances of the subsystem are active and you request a long subsystem status or subserver status. If you specify a null *SubsystemPID* parameter, you must specify a *SubsystemName* parameter.
StatusType	Specifies **LONGSTAT** for long status or **SHORTSTAT** for short status.
StatusFrom	Specifies whether status errors and messages are to be printed to standard output or just returned to the caller. When the *StatusFrom* parameter is **SSHELL**, the errors are printed to standard output.

Item	Description
StatusText	Allocates memory for the printable text and sets the *StatusText* parameter to point to this memory. After it prints the text, the calling process must free the memory allocated for this buffer.
Continued	Specifies whether this call to the **srcsbuf** subroutine is a continuation of a status request. If the *Continued* parameter is set to **NEWREQUEST**, a request for status is sent and the **srcsbuf** subroutine then waits for another. On return, the **srcsbuf** subroutine is updated to the new continuation indicator from the reply packet and the *Continued* parameter is set to **END** or **STATCONTINUED** by the subsystem. If the *Continued* parameter is set to something other than **END**, this field must remain equal to that value; otherwise, this function will not be able to receive any more packets for the original status request. The calling process should not set the value of the *Continued* parameter to a value other than **NEWREQUEST**. The *Continued* parameter should not be changed while more responses are expected.

Return Values

If the **srcsbuf** subroutine succeeds, it returns the size (in bytes) of printable text pointed to by the *StatusText* parameter.

Error Codes

The **srcsbuf** subroutine fails if one or more of the following are true:

Item	Description
SRC_BADSOCK	The request could not be passed to the subsystem because of some socket failure.
SRC_CONT	The subsystem uses signals. The request cannot complete.
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	The *Continued* parameter was not set to **NEWREQUEST**, and no continuation is currently active.
SRC_NORPLY	The request timed out waiting for a response.
SRC_NSVR	The subsystem is not active.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_STPG	The request was not passed to the subsystem. The subsystem is stopping.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.
SRC_WICH	There are multiple instances of the subsystem active.

Examples

1. To get the status of a subsystem, enter:

```
char *status;
int continued=NEWREQUEST;
int rc;

do {
  rc=srcsbuf("MaryC", SUBSYSTEM, "srctest", "", 0,
     SHORTSTAT, SSHELL, &status, continued);
   if (status!=0)
    {
        printf(status);
        free(status);
        status=0;
    }
}  while (rc>0);
```

This gets short status of the `srctest` subsystem on the `MaryC` machine and prints the formatted status to standard output.

2. To get the status of a subserver, enter:

```
char *status;
int continued=NEWREQUEST;
int rc;

do {
  rc=srcsbuf("", 12345, "srctest", "", 0,
     LONGSTAT, SSHELL, &status, continued);
   if (status!=0)
    {
        printf(status);
        free(status);
        status=0;
    }
}  while (rc>0);
```

This gets long status for a specific subserver belonging to subsystem `srctest`. The subserver is the one having code point 12345. This request is processed on the local machine. The formatted status is printed to standard output.

Files

Item	Description
/etc/services	Defines sockets and protocols used for Internet services.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

srcsbuf_r Subroutine

Purpose

Gets status for a subserver or a subsystem and returns status text to be printed.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <spc.h>

int srcsbuf_r(Host, Type, SubsystemName, SubserverObject, SubsystemPID,
StatusType, StatusFrom, StatusText, Continued, SRCHandle)
```

```
char * Host, * SubsystemName;
char * SubserverObject, ** StatusText;
short  Type,  StatusType;
pid_t  SubsystemPID;
int  StatusFrom, * Continued;
char ** SRCHandle;
```

Description

The **srcsbuf_r** subroutine gets the status of a subserver or subsystem and returns printable text for the status in the address pointed to by the *StatusText* parameter. The **srcsbuf_r** subroutine supports all the functions of the **srcbuf** subroutine except the *StatusFrom* parameter.

When the *StatusType* parameter is **SHORTSTAT** and the *Type* parameter is **SUBSYSTEM**, the **srcstat_r** subroutine is called to get the status of one or more subsystems. When the *StatusType* parameter is **LONGSTAT** and the *Type* parameter is **SUBSYSTEM**, the **srcrsqt_r** subroutine is called to get the long status of one subsystem. When the *Type* parameter is not **SUBSYSTEM**, the **srcsrqt_r** subroutine is called to get the long or short status of a subserver.

This routine is threadsafe and reentrant.

Parameters

Item	Description
Host	Specifies the foreign host on which this status action is requested. If the host is null, the status request is sent to the System Resource Controller (SRC) on the local host.
Type	Specifies whether the status request applies to the subsystem or subserver. If the *Type* parameter is set to **SUBSYSTEM**, the status request is for a subsystem. If not, the status request is for a subserver and the *Type* parameter is a subserver code point.
SubsystemName	Specifies the name of the subsystem on which to get status. To get the status of all subsystems, use the **SRCALLSUBSYS** constant. To get the status of a group of subsystems, the *SubsystemName* parameter must start with the **SRCGROUP** constant, followed by the name of the group for which you want status appended. If you specify a null *SubsystemName* parameter, you must specify a *SubsystemPID* parameter.
SubserverObject	Specifies a subserver object. The *SubserverObject* parameter modifies the *Type* parameter. The *SubserverObject* parameter is ignored if the *Type* parameter is set to **SUBSYSTEM**. The use of the *SubserverObject* parameter is determined by the subsystem and the caller. This parameter will be placed in the `objname` field of the **subreq** structure that is passed to the subsystem.
SubsystemPID	Specifies the process ID of the subsystem on which to get status, as returned by the **srcstrt** subroutine. You must specify the *SubsystemPID* parameter if multiple instances of the subsystem are active and you request a long subsystem status or subserver status. If you specify a null *SubsystemPID* parameter, you must specify a *SubsystemName* parameter.
StatusType	Specifies **LONGSTAT** for long status or **SHORTSTAT** for short status.
StatusFrom	Specifies whether status errors and messages are to be printed to standard output or just returned to the caller. When the *StatusFrom* parameter is **SSHELL**, the errors are printed to standard output. The **SSHELL** value is not recommended in a multithreaded environment since error messages to standard output may be interleaved in an unexpected manner.

Item	Description
StatusText	Allocates memory for the printable text and sets the *StatusText* parameter to point to this memory. After it prints the text, the calling process must free the memory allocated for this buffer.
Continued	Specifies whether this call to the **srcsbuf_r** subroutine is a continuation of a status request. If the *Continued* parameter is set to **NEWREQUEST**, a request for status is sent and the **srcsbuf_r** subroutine then waits for a reply. On return from the **srcsbuf_r** subroutine, the *Continued* parameter is updated to the new continuation indicator from the reply packet. The continuation indicator in the reply packet will be set to **END** or **STATCONTINUED** by the subsystem. If the *Continued* parameter is set to something other than **END**, the caller should not change that value; otherwise, this function will not be able to receive any more packets for the original status request. The calling process should not set the value of the *Continued* parameter to a value other than **NEWREQUEST**. In normal processing, the *Continued* parameter should not be changed while more responses are expected. The caller must continue to call the **srcsbuf_r** subroutine until **END** is received. As an alternative, call the **srcsbuf_r** subroutine with **Continued=SRC_CLOSE** to discard the remaining data, close the socket, and free the internal buffers.
SRCHandle	Identifies a request and its associated responses. Set to NULL by the caller for a **NEWREQUEST**. The **srcsbuf_r** subroutine saves a value in *SRCHandle* to allow **srcsbuf_r** continuation calls to use the same socket and internal buffers. The *SRCHandle* parameter should not be changed by the caller except for **NEWREQUEST**s.

Return Values

If the **srcsbuf_r** subroutine succeeds, it returns the size (in bytes) of printable text pointed to by the *StatusText* parameter.

Error Codes

The **srcsbuf_r** subroutine fails and returns the corresponding error code if one of the following error conditions is detected:

Item	Description
SRC_BADSOCK	The request could not be passed to the subsystem because of some socket failure.
SRC_CONT	The subsystem uses signals. The request cannot complete.
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	The *Continued* parameter was not set to **NEWREQUEST**, and no continuation is currently active.
SRC_NORPLY	The request timed out waiting for a response.
SRC_NSVR	The subsystem is not active.
SRC_SOCK	There is a problem with SRC socket communications.

Item	Description
SRC_STPG	The request was not passed to the subsystem. The subsystem is stopping.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.
SRC_WICH	There are multiple instances of the subsystem active.

Examples

1. To get the status of a subsystem, enter:

```
char *status;
int continued=NEWREQUEST;
int rc;
char *handle

do {
  rc=srcsbuf_r("MaryC", SUBSYSTEM, "srctest", "", 0,
     SHORTSTAT, SDAEMON, &status, continued, &handle);
   if (status!=0)
    {
        printf(status);
        free(status);
        status=0;
    }
}  while (rc>0);
if (rc<0)
{
   ...handle error from srcsbuf_r...
}
```

   This gets short status of the `srctest` subsystem on the `MaryC` machine and prints the formatted status to standard output.

   > **Caution:** In a multithreaded environment, the caller must manage the sharing of standard output between threads. Set the *StatusFrom* parameter to SDAEMON to prevent unexpected error messages from being printed to standard output.

2. To get the status of a subserver, enter:

```
char *status;
int continued=NEWREQUEST;
int rc;
char *handle

do {
  rc=srcsbuf_r("", 12345, "srctest", "", 0,
     LONGSTAT, SDAEMON, &status, continued, &handle);
   if (status!=0)
    {
        printf(status);
        free(status);
        status=0;
    }
}  while (rc>0);
if (rc<0)
{
   ...handle error from srcsbuf_r...
}
```

   This gets long status for a specific subserver belonging to subsystem `srctest`. The subserver is the one having code point 12345. This request is processed on the local machine. The formatted status is printed to standard output.

   

   CAUTION: In a multithreaded environment, the caller must manage the sharing of standard output between threads. Set the *StatusFrom* parameter to SDAEMON to prevent unexpected error messages from being printed to standard output.

srcsrpy Subroutine

Purpose

Sends a reply to a request from the System Resource Controller (SRC) back to the client process.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <spc.h>
```

int srcsrpy (*SRChdr*, *PPacket*, *PPacketSize*, *Continued*)

```
struct srchdr *SRChdr;
char *PPacket;
int PPacketSize;
ushort Continued;
```

Description

The **srcsrpy** subroutine returns a subsystem reply to a System Resource Controller (SRC) subsystem request. The format and content of the reply are determined by the subsystem and the requester, but must start with a **srchdr** structure. This structure and all others required for subsystem communication with the SRC are defined in the **/usr/include/spc.h** file. The subsystem must reply with a pre-defined format and content for the following requests: **START**, **STOP**, **STATUS**, **REFRESH**, and **TRACE**. The **START**, **STOP**, **REFRESH**, and **TRACE** requests must be answered with a **srcrep** structure. The **STATUS** request must be answered with a reply in the form of a **statbuf** structure.

Note: The **srcsrpy** subroutine creates its own socket to send the subsystem reply packets.

Parameters

Item	Description
SRChdr	Points to the reply address buffer as returned by the **srcrrqs** subroutine.
PPacket	Points to the reply packet. The first element of the reply packet is a **srchdr** structure. The `cont` element of the *PPacket*->**srchdr** structure is modified on returning from the **srcsrpy** subroutine. The second element of the reply packet should be a **svrreply** structure, an array of **statcode** structures, or another format upon which the subsystem and the requester have agreed.
PPacketSize	Specifies the number of bytes in the reply packet pointed to by the *PPacket* parameter. The *PPacketSize* parameter may be the size of a **short**, or it may be between the size of a **srchdr** structure and the **SRCPKTMAX** value, which is defined in the **spc.h** file.

Item	Description
Continued	Indicates whether this reply is to be continued. If the *Continued* parameter is set to the constant **END**, no more reply packets are sent for this request. If the *Continued* parameter is set to **CONTINUED**, the second element of what is indicated by the *PPacket* parameter must be a **svrreply** structure, since the `rtnmsg` element of the **svrreply** structure is printed to standard output. For a status reply, the *Continued* parameter is set to **STATCONTINUED**, and the second element of what is pointed to by the *PPacket* parameter must be an array of **statcode** structures. If a STOP subsystem request is received, only one reply packet can be sent and the *Continued* parameter must be set to **END**. Other types of continuations, as determined by the subsystem and the requester, must be defined using positive values for the *Continued* parameter. Values other than the following must be used: **0** **END** **1** **CONTINUED** **2** **STATCONTINUED**

Return Values

If the **srcsrpy** subroutine succeeds, it returns the value **SRC_OK**.

Error Codes

The **srcsrpy** subroutine fails if one or both of the following are true:

Item	Description
SRC_SOCK	There is a problem with SRC socket communications.
SRC_REPLYSZ	SRC reply size is invalid.

Examples

1. To send a **STOP** subsystem reply, enter:

```
struct srcrep return_packet;
struct srchdr *srchdr;

bzero(&return_packet,sizeof(return_packet));
return_packet.svrreply.rtncode=SRC_OK;
strcpy(return_packet.svrreply,"srctest");

srcsrpy(srchdr,return_packet,sizeof(return_packet),END);
```

   This entry sends a message that the subsystem `srctest` is stopping successfully.

2. To send a **START** subserver reply, enter:

```
struct srcrep return_packet;
struct srchdr *srchdr;

bzero(&return_packet,sizeof(return_packet));
return_packet.svrreply.rtncode=SRC_SUBMSG;
strcpy(return_packet.svrreply,objname,"mysubserver");
strcpy(return_packet.svrreply,objtext,"The subserver,\
mysubserver, has been started");

srcsrpy(srchdr,return_packet,sizeof(return_packet),END);
```

   The resulting message indicates that the start subserver request was successful.

3. To send a status reply, enter:

```
int rc;
struct sockaddr addr;
int addrsz;
struct srcreq packet;
struct
{
   struct srchdr srchdr;
   struct statcode statcode[10];
} status;
struct srchdr *srchdr;
struct srcreq packet;
   .
   .
   .
/* grab the reply information from the SRC packet */
srchdr=srcrrqs(&packet);
bzero(&status.statcode[0].objname,

/* get SRC status header */
srcstathdr(status.statcode[0].objname,
  status.statcode[0].objtext);
   .
   .
   .
/* send status packet(s) */
srcsrpy(srchdr,&status,sizeof(status),STATCONTINUED);
   .
   .
   .
srcsrpy(srchdr,&status,sizeof(status),STATCONTINUED);

/* send final packet */
srcsrpy(srchdr,&status,sizeof(struct srchdr),END);
```

This entry sends several status packets.

Files

Item	Description
/dev/.SRC-unix	Specifies the location for temporary socket files.

srcsrqt Subroutine

Purpose

Sends a request to a subsystem.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

#include <spc.h> srcsrqt(*Host***, *SubsystemName*, *SubsystemPID*, *RequestLength*, *SubsystemRequest*, *ReplyLength*, *ReplyBuffer*, *StartItAlso*, *Continued*)**

char * *Host*, ***** *SubsystemName*;

char * *SubsystemRequest*, ***** *ReplyBuffer*;

int SubsystemPID, *StartItAlso*, ***** *Continued*;

short *RequestLength*, * *ReplyLength*;

Description

The **srcsrqt** subroutine sends a request to a subsystem, waits for a response, and returns one or more replies to the caller. The format of the request and the reply is determined by the caller and the subsystem.

Note: The **srcsrqt** subroutine creates its own socket to send a request to the subsystem. The socket that this function opens remains open until an error or an end packet is received.

Two types of continuation are returned by the **srcsrqt** subroutine:

Item	Description
No continuation	*ReplyBuffer*->`srchdr.continued` is set to the **END** constant.
Reply continuation	*ReplyBuffer*->`srchdr.continued` is not set to the **END** constant, but to a positive value agreed upon by the calling process and the subsystem. The packet is returned to the caller.

Parameters

Item	Description
SubsystemPID	The process ID of the subsystem.
Host	Specifies the foreign host on which this subsystem request is to be sent. If the host is null, the request is sent to the subsystem on the local host. The local user must be running as "root". The remote system must be configured to accept remote System Resource Controller requests. That is, the **srcmstr** daemon (see **/etc/inittab**) must be started with the **-r** flag and the **/etc/hosts.equiv** or **.rhosts** file must be configured to allow remote requests.
SubsystemName	Specifies the name of the subsystem to which this request is to be sent. You must specify a *SubsystemName* if you do not specify a *SubsystemPID.*
RequestLength	Specifies the length, in bytes, of the request to be sent to the subsystem. The maximum value in bytes for this parameter is 2000 bytes.
SubsystemRequest	Points to the subsystem request packet.
ReplyLength	Specifies the maximum length, in bytes, of the reply to be received from the subsystem. On return from the **srcsrqt** subroutine, the *ReplyLength* parameter is set to the actual length of the subsystem reply packet.
ReplyBuffer	Points to a buffer for the receipt of the reply packet from the subsystem.
StartItAlso	Specifies whether the subsystem should be started if it is nonactive. When nonzero, the System Resource Controller (SRC) attempts to start a nonactive subsystem, and then passes the request to the subsystem.
Continued	Specifies whether this call to the **srcsrqt** subroutine is a continuation of a previous request. If the *Continued* parameter is set to **NEWREQUEST**, a request for it is sent to the subsystem and the subsystem is notified that another response is expected. The calling process should never set *Continued* to any value other than **NEWREQUEST**. The last response from the subsystem will set *Continued* to **END**.

Return Values

If the **srcsrqt** subroutine is successful, the value **SRC_OK** is returned.

Error Codes

The **srcsrqt** subroutine fails if one or more of the following are true:

Item	Description
SRC_BADSOCK	The request could not be passed to the subsystem because of a socket failure.
SRC_CONT	The subsystem uses signals. The request cannot complete.
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	The *Continued* parameter was not set to **NEWREQUEST**, and no continuation is currently active.
SRC_NORPLY	The request timed out waiting for a response.
SRC_NSVR	The subsystem is not active.
SRC_REQLEN2BIG	The *RequestLength* is greater than the maximum 2000 bytes.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_STPG	The request was not passed to the subsystem. The subsystem is stopping.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.

Examples

1. To request long subsystem status, enter:

```
int cont=NEWREQUEST;
int rc;
short replen;
short reqlen;
struct
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;

subreq.action=STATUS;
subreq.object=SUBSYSTEM;
subreq.parm1=LONGSTAT;
strcpy(subreq.objname,"srctest");
replen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt("MaryC", "srctest", 0, reqlen, &subreq, &replen,
&statbuf, SRC_NO, &cont);
```

   This entry gets long status of the subsystem `srctest` on the `MaryC` machine. The subsystem keeps sending status packets until `statbuf.srchdr.cont=END`.

2. To start a subserver, enter:

```
int cont=NEWREQUEST;
int rc;
short replen;
short reqlen;
struct
```

```
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;

subreq.action=START;
subreq.object=1234;
replen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt("", "", 987, reqlen, &subreq, &replen, &statbuf,
SRC_NO, &cont);
```

This entry starts the subserver with the code point of 1234, but only if the subsystem is already active.

3. To start a subserver and a subsystem, enter:

```
int cont=NEWREQUEST;
int rc;
short replen;
short reqlen;
struct
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;
subreq.action=START;
subreq.object=1234;
replen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt("", "", 987, reqlen, &subreq, &replen, &statbuf, SRC_YES, &cont);
```

This entry starts the subserver with the code point of 1234. If the subsystem to which this subserver belongs is not active, the subsystem is started.

Files

Item	Description
/etc/services	Defines sockets and protocols used for Internet services.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

srcsrqt_r Subroutine

Purpose

Sends a request to a subsystem.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <spc.h>

srcsrqt_r(Host, SubsystemName, SubsystemPID, RequestLength,
             SubsystemRequest, ReplyLength, ReplyBuffer, StartItAlso,
             Continued, SRCHandle)
char * Host, * SubsystemName;
char * SubsystemRequest, * ReplyBuffer;
pid_t  SubsystemPID,
```

```
int  StartItAlso, * Continued;
short  RequestLength, * ReplyLength;
char ** SRCHandle;
```

Description

The **srcsrqt_r** subroutine sends a request to a subsystem, waits for a response and returns one or more replies to the caller. The format of the request and the reply is determined by the caller and the subsystem.

Note: For each **NEWREQUEST**, the **srcsrqt_r** subroutine creates its own socket to send a request to the subsystem. The socket that this function opens remains open until an error or an end packet is received.

This system is threadsafe and reentrant.

Two types of continuation are returned by the **srcsrqt_r** subroutine:

Item	Description
No continuation	*ReplyBuffer*->`srchdr.continued` is set to the **END** constant.
Reply continuation	*ReplyBuffer*->`srchdr.continued` is not set to the **END** constant, but to a positive value agreed upon by the calling process and the subsystem. The packet is returned to the caller.

Parameters

Item	Description
SubsystemPID	The process ID of the subsystem.
Host	Specifies the foreign host on which this subsystem request is to be sent. If the host is null, the request is sent to the subsystem on the local host.
SubsystemName	Specifies the name of the subsystem to which this request is to be sent. You must specify a *SubsystemName* if you do not specify a *SubsystemPID.*
RequestLength	Specifies the length, in bytes, of the request to be sent to the subsystem. The maximum length is 2000 bytes.
SubsystemRequest	Points to the subsystem request packet.
ReplyLength	Specifies the maximum length, in bytes, of the reply to be received from the subsystem. On return from the **srcsrqt** subroutine, the *ReplyLength* parameter is set to the actual length of the subsystem reply packet.
ReplyBuffer	Points to a buffer for the receipt of the reply packet from the subsystem.
StartItAlso	Specifies whether the subsystem should be started if it is nonactive. When nonzero, the System Resource Controller (SRC) attempts to start a nonactive subsystem, and then passes the request to the subsystem.
Continued	Specifies whether this call to the **srcsrqt** subroutine is a continuation of a previous request. If the *Continued* parameter is set to **NEWREQUEST**, a request for it is sent to the subsystem and the subsystem is notified that a response is expected. Under normal circumstances, the calling process should never set *Continued* to any value other than **NEWREQUEST**. The last response from the subsystem will set *Continued* to **END**. The caller must continue to call the **srcsrqt_r** subroutine until **END** is received. Otherwise, the socket will not be closed and the internal buffers freed. As an alternative, set *Continued=SRC_CLOSE* to discard the remaining data, close the socket, and free the internal buffers.

Item	Description
SRCHandle	Identifies a request and its associated responses. Set to NULL by the caller for a **NEWREQUEST**. The **srcsrqt_r** subroutine saves a value in *SRCHandle* to allow **srcsrqt_r** continuation calls to use the same socket and internal buffers. The *SRCHandle* parameter should not be changed by the caller except for **NEWREQUEST**s.

Return Values

If the **srcsrqt_r** subroutine is successful, the value **SRC_OK** is returned.

Error Codes

The **srcsrqt_r** subroutine fails and returns the corresponding error code if one of the following error conditions is detected:

Item	Description
SRC_BADSOCK	The request could not be passed to the subsystem because of a socket failure.
SRC_CONT	The subsystem uses signals. The request cannot complete.
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	The *Continued* parameter was not set to **NEWREQUEST**, and no continuation is currently active.
SRC_NORPLY	The request timed out waiting for a response.
SRC_NSVR	The subsystem is not active.
SRC_REQLEN2BIG	The *RequestLength* is greater than the maximum 2000 bytes.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_STPG	The request was not passed to the subsystem. The subsystem is stopping.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.

Examples

1. To request long subsystem status, enter:

```
int cont=NEWREQUEST;
int rc;
short replen;
short reqlen;
char *handle;
struct
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;

subreq.action=STATUS;
```

```
subreq.object=SUBSYSTEM;
subreq.parm1=LONGSTAT;
strcpy(subreq.objname,"srctest");
replen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt_r("MaryC", "srctest", 0, reqlen, &subreq, &replen,
&statbuf, SRC_NO, &cont, &handle);
```

This entry gets long status of the subsystem `srctest` on the `MaryC` machine. The subsystem keeps sending status packets until `statbuf.srchdr.cont=END`.

2. To start a subserver, enter:

```
int cont=NEWREQUEST;
int rc;
short replen;
short reqlen;
struct
char *handle;
struct
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;

subreq.action=START;
subreq.object=1234;
replen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt_r("", "", 987, reqlen, &subreq, &replen, &statbuf,
SRC_NO, &cont, &handle);
```

This entry starts the subserver with the code point of 1234, but only if the subsystem is already active.

3. To start a subserver and a subsystem, enter:

```
int cont=NEWREQUEST;
int rc;
short replen;
short reqlen;
char *handle;
struct
{
    struct srchdr srchdr;
    struct statcode statcode[20];
} statbuf;
struct subreq subreq;
subreq.action=START;
subreq.object=1234;
replen=sizeof(statbuf);
reqlen=sizeof(subreq);
rc=srcsrqt("", "", 987, reqlen, &subreq, &replen, &statbuf, SRC_YES, &cont, &handle);
```

This entry starts the subserver with the code point of `1234`. If the subsystem to which this subserver belongs is not active, the subsystem is started.

Files

Item	Description
/etc/services	Defines sockets and protocols used for Internet services.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

srcstat Subroutine

Purpose

Gets short status on one or more subsystems.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <spc.h>
```

```
int srcstat(Host,
SubsystemName,SubsystemPID, ReplyLength, StatusReply,Continued)
char * Host, * SubsystemName;
int  SubsystemPID * Continued;
short * ReplyLength;
void * StatusReply;
```

Description

The **srcstat** subroutine sends a short status request to the System Resource Controller (SRC) and returns status for one or more subsystems to the caller.

Parameters

Item	Description
Host	Specifies the foreign host on which this status action is requested. If the host is null, the status request is sent to the SRC on the local host. The local user must be running as "root". The remote system must be configured to accept remote System Resource Controller requests. That is, the **srcmstr** daemon (see **/etc/inittab**) must be started with the **-r** flag and the **/etc/hosts.equiv** or **.rhosts** file must be configured to allow remote requests.
SubsystemName	Specifies the name of the subsystem on which to get short status. To get status of all subsystems, use the **SRCALLSUBSYS** constant. To get status of a group of subsystems, the *SubsystemName* parameter must start with the **SRCGROUP** constant, followed by the name of the group for which you want status appended. If you specify a null *SubsystemName* parameter, you must specify a *SubsystemPID* parameter.
SubsystemPID	Specifies the PID of the subsystem on which to get status as returned by the **srcstat** subroutine. You must specify the *SubsystemPID* parameter if multiple instances of the subsystem are active and you request a long subsystem status or subserver status. If you specify a null *SubsystemPID* parameter, you must specify a *SubsystemName* parameter.
ReplyLength	Specifies size of a **srchdr** structure plus the number of **statcode** structures times the size of one **statcode** structure. On return from the **srcstat** subroutine, this value is updated.
StatusReply	Specifies a pointer to a structure containing first element as struct **srchdr** and secondary element as struct **statcode** (both defined in **spc.h** file) array that receives the status reply for the requested subsystem. The first element of the returned statcode array contains the status title line. The number of statcode structures array items depends on the number of subsystems user queried.
Continued	Specifies whether this call to the **srcstat** subroutine is a continuation of a previous status request. If the *Continued* parameter is set to **NEWREQUEST**, a request for short subsystem status is sent to the SRC and **srcstat** waits for the first status response. The calling process should never set *Continued* to a value other than **NEWREQUEST**. The last response for the SRC sets *Continued* to **END**.

Return Values

If the **srcstat** subroutine succeeds, it returns a value of 0. An error code is returned if the subroutine is unsuccessful.

Error Codes

The **srcstat** subroutine fails if one or more of the following are true:

Item	Description
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	*Continued* was not set to **NEWREQUEST** and no continuation is currently active.
SRC_NORPLY	The request timed out waiting for a response.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.

Examples

1. To request the status of a subsystem, enter:

```
intcont=NEWREQUEST;
struct {
           struct srchdr srchdr
           struct statcode statcode[6];
} status;
short replen=sizeof(status);

srcstat("MaryC","srctest",0,&replen,&status,&cont);
```

   This entry requests short status of all instances of the subsystem `srctest` on the `MaryC` machine.

2. To request the status of all subsystems, enter:

```
int cont=NEWREQUEST;
struct {
    struct srchdr srchdr;
    struct statcode statcode[80];
} status;
short replen=sizeof(status);

srcstat("",SRCALLSUBSYS,0,&replen,&status,&cont);
```

   This entry requests short status of all subsystems on the local machine.

3. To request the status for a group of subsystems, enter:

```
int cont=NEWREQUEST;
struct struct {
    struct srchdr srchdr;
struct statcode statcode[30];
} status;
short replen=sizeof(status), rep_num;
char subsysname[30];

strcpy(subsysname,SRCGROUP);
strcat(subsysname,"tcpip");
```

```
srcstat("",subsysname,0,&replen,&status, &cont);

rep_num = (replen - sizeof(strcut srchdr)) / sizeof(strcut statcode);

for (i = 0; i < rep_num; i++)
    printf("objtype %d status %d objname %s objtext %s\n",
            status.statcode[i].objtype, status.statcode[i].status,
            status.statcode[i].objname, status.statcode[i].objtext);
```

This entry requests short status of all members of the subsystem group `tcpip` on the local machine , and displays the query results on **stdout**.

Files

Item	Description
/etc/services	Defines the sockets and protocols used for Internet services.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

srcstat_r Subroutine

Purpose

Gets short status on a subsystem.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <spc.h>

int srcstat_r(Host, SubsystemName, SubsystemPID, ReplyLength,
                StatusReply, Continued, SRCHandle)
char * Host, * SubsystemName;
pid_t  SubsystemPID;
int * Continued;
short * ReplyLength;
struct statrep * StatusReply;
char ** SRCHandle;
```

Description

The **srcstat_r** subroutine sends a short status request to the System Resource Controller (SRC) and returns status for one or more subsystems to the caller. This subroutine is threadsafe and reentrant.

Parameters

Item	Description
Host	Specifies the foreign host on which this status action is requested. If the host is null, the status request is sent to the SRC on the local host.

Item	Description
SubsystemName	Specifies the name of the subsystem on which to get short status. To get status of all subsystems, use the **SRCALLSUBSYS** constant. To get status of a group of subsystems, the *SubsystemName* parameter must start with the **SRCGROUP** constant, followed by the name of the group for which you want status appended. If you specify a null *SubsystemName* parameter, you must specify a *SubsystemPID* parameter.
SubsystemPID	Specifies the PID of the subsystem on which to get status as returned by the **srcstat_r** subroutine. You must specify the *SubsystemPID* parameter if multiple instances of the subsystem are active and you request a long subsystem status or subserver status. If you specify a null *SubsystemPID* parameter, you must specify a *SubsystemName* parameter.
ReplyLength	Specifies size of a **srchdr** structure plus the number of **statcode** structures times the size of one **statcode** structure. On return from the **srcstat_r** subroutine, this value is updated.
StatusReply	Specifies a pointer to a **statrep** code structure containing a **statcode** array that receives the status reply for the requested subsystem. The first element of the returned **statcode** array contains the status title line. The **statcode** structure is defined in the **spc.h** file.
Continued	Specifies whether this call to the **srcstat_r** subroutine is a continuation of a previous status request. If the *Continued* parameter is set to **NEWREQUEST**, a request for short subsystem status is sent to the SRC and **srcstat_r** waits for the first status response. During **NEWREQUEST** processing, the **srcstat_r** subroutine opens a socket, mallocs internal buffers, and saves a value in *SRCHandle*. In normal circumstances, the calling process should never set *Continued* to a value other than **NEWREQUEST**. When the **srcstat_r** subroutine returns with *Continued=STATCONTINUED*, call **srcstat_r** without changing the *Continued* and *SRCHandle* parameters to receive additional data. The last response from the SRC sets *Continued* to **END**. The caller must continue to call **srcstat_r** until **END** is received. Otherwise, the socket will not be closed and the internal buffers freed. As an alternative, call **srcstat_r** with *Continued=STATCONTINUED* to discard the remaining data, close the socket, and free the internal buffers.
SRCHandle	Identifies a request and its associated responses. Set to NULL by the caller for a **NEWREQUEST**. The **srcstat_r** subroutine saves a value in *SRCHandle* to allow subsequent **srcstat_r** calls to use the same socket and internal buffers. The *SRCHandle* parameter should not be changed by the caller except for **NEWREQUEST**s.

Return Values

If the **srcstat_r** subroutine succeeds, it returns a value of 0. An error code is returned if the subroutine is unsuccessful.

Error Codes

The **srcstat_r** subroutine fails and returns the corresponding error code if one of the following error conditions is detected:

Item	Description
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.

Item	Description
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NOCONTINUE	*Continued* was not set to **NEWREQUEST** and no continuation is currently active.
SRC_NORPLY	The request timed out waiting for a response.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_UDP	The SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.

Examples

1. To request the status of a subsystem, enter:

```
int cont=NEWREQUEST;
struct statcode statcode[20];
short replen=sizeof(statcode);
char *handle;

srcstat_r("MaryC","srctest",0,&replen,statcode, &cont, &handle);
```

   This entry requests short status of all instances of the subsystem `srctest` on the `MaryC` machine.

2. To request the status of all subsystems, enter:

```
int cont=NEWREQUEST;
struct statcode statcode[20];
short replen=sizeof(statcode);
char *handle;

srcstat_r("",SRCALLSUBSYS,0,&replen,statcode, &cont, &handle);
```

   This entry requests short status of all subsystems on the local machine.

3. To request the status for a group of subsystems, enter:

```
int cont=NEWREQUEST;
struct statcode statcode[20];
short replen=sizeof(statcode);
char subsysname[30];
char *handle;

strcpy(subsysname,SRCGROUP);
strcat(subsysname,"tcpip");
srcstat_r("",subsysname,0,&replen,statcode, &cont, &handle);
```

   This entry requests short status of all members of the subsystem group `tcpip` on the local machine.

Files

Item	Description
/etc/services	Defines the sockets and protocols used for Internet services.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

srcstathdr Subroutine

Purpose

Gets the title line of the System Resource Controller (SRC) status text.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
void srcstathdr ( Title1,  Title2)
char *Title1, *Title2;
```

Description

The **srcstathdr** subroutine retrieves the title line, or header, of the SRC status text.

Parameters

Item	Description
Title1	Specifies the `objname` field of a **statcode** structure. The subsystem name title is placed here.
Title2	Specifies the `objtext` field of a **statcode** structure. The remaining titles are placed here.

Return Values

The subsystem name title is returned in the *Title1* parameter. The remaining titles are returned in the *Title2* parameter.

srcstattxt Subroutine

Purpose

Gets the System Resource Controller (SRC) status text representation for a status code.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
char *srcstattxt ( StatusCode)
short StatusCode;
```

Description

The **srcstattxt** subroutine, given an SRC status code, gets the text representation and returns a pointer to this text.

Parameters

Item	Description
StatusCode	Specifies an SRC status code to be translated into meaningful text.

Return Values

The **srcstattxt** subroutine returns a pointer to the text representation of a status code.

srcstattxt_r Subroutine

Purpose

Gets the status text representation for an SRC status code.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <spc.h>

char *srcstattxt_r (StatusCode, Text)
short StatusCode;
char *Text;
```

Description

The **srcstattxt_r** subroutine, given an SRC status code, gets the text representation and returns it in a caller-supplied buffer. This routine is threadsafe and reentrant.

Parameters

Item	Description
StatusCode	Specifies an SRC status code to be translated into meaningful text.
Text	Points to a caller-supplied buffer where the text will be returned. If the length of the text is unknown, the maximum text length can be used when allocating the buffer. The maximum text length is **SRC_STAT_MAX** in **/usr/include/spc.h** (64 bytes).

Return Values

Upon successful completion, the **srcstattxt_r** subroutine returns the address of the caller-supplied buffer. Otherwise, no text is returned and the subroutine returns NULL.

srcstop Subroutine

Purpose

Stops a System Resource Controller (SRC) subsystem.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include <spc.h>
```

```
srcstop(Host, SubsystemName, SubsystemPID, StopType)
srcstop(ReplyLength, ServerReply, StopFrom)
char * Host, * SubsystemName;
int  SubsystemPID,  StopFrom;
short  StopType, * ReplyLength;
struct srcrep * ServerReply;
```

Description

The **srcstop** subroutine sends a stop subsystem request to a subsystem and waits for a stop reply from the System Resource Controller (SRC) or the subsystem. The **srcstop** subroutine can only stop a subsystem that was started by the SRC.

Parameters

Item	Description
Host	Specifies the foreign host on which this stop action is requested. If the host is the null value, the request is sent to the SRC on the local host. The local user must be running as "root". The remote system must be configured to accept remote System Resource Controller requests. That is, the **srcmstr** daemon (see **/etc/inittab**) must be started with the **-r** flag and the **/etc/hosts.equiv** or **.rhosts** file must be configured to allow remote requests.
SubsystemName	Specifies the name of the subsystem to stop.
SubsystemPID	Specifies the process ID of the system to stop as returned by the **srcstrt** subroutine. If you specify a null *SubsystemPID* parameter, you must specify a *SubsystemName* parameter.
StopType	Specifies the type of stop requested of the subsystem. If this parameter is null, a normal stop is assumed. The *StopType* parameter must be one of the following values: **CANCEL** Requires a quick stop of the subsystem. The subsystem is sent a **SIGTERM** signal. After the wait time defined in the subsystem object, the SRC issues a **SIGKILL** signal to the subsystem. This waiting period allows the subsystem to clean up all its resources and terminate. The stop reply is returned by the SRC. **FORCE** Requests a quick stop of the subsystem and all its subservers. The stop reply is returned by the SRC for subsystems that use signals and by the subsystem for other communication types. **NORMAL** Requests the subsystem to terminate after all current subsystem activity has completed. The stop reply is returned by the SRC for subsystems that use signals and by the subsystem for other communication types.
ReplyLength	Specifies the maximum length, in bytes, of the stop reply. On return from the **srcstop** subroutine, this field is set to the actual length of the subsystem reply packet received.
ServerReply	Points to an **svrreply** structure that will receive the subsystem stop reply.
StopFrom	Specifies whether the **srcstop** subroutine is to display stop results to standard output. If the *StopFrom* parameter is set to **SSHELL**, the stop results are displayed to standard output and the **srcstop** subroutine returns successfully. If the *StopFrom* parameter is set to **SDAEMON**, the stop results are not displayed to standard output, but are passed back to the caller.

Return Values

Upon successful completion, the **srcstop** subroutine returns **SRC_OK** or **SRC_STPOK**.

Error Codes

The **srcstop** subroutine fails if one or more of the following are true:

Item	Description
SRC_BADFSIG	The stop force signal is an invalid signal.
SRC_BADNSIG	The stop normal signal is an invalid signal.
SRC_BADSOCK	The stop request could not be passed to the subsystem on its communication socket.
SRC_DMNA	The SRC daemon is not active.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_NORPLY	The request timed out waiting for a response.
SRC_NOTROOT	The SRC daemon is not running as root.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_STPG	The request was not passed to the subsystem. The subsystem is stopping.
SRC_SVND	The subsystem is unknown to the SRC daemon.
SRC_UDP	The remote SRC port is not defined in the **/etc/services** file.
SRC_UHOST	The foreign host is not known.
SRC_PARM	Invalid parameter passed.

Examples

1. To stop all instances of a subsystem, enter:

```
int rc;
struct svrreply svrreply;
short replen=sizeof(svrreply);

rc=srcstop("MaryC","srctest",0,FORCE,&replen,&svrreply,SDAEMON);
```

   This request stops a subsystem with a stop type of FORCE for all instances of the subsystem `srctest` on the `MaryC` machine and does not print a message to standard output about the status of the stop.

2. To stop a single instance of a subsystem, enter:

```
struct svrreply svrreply;
short replen=sizeof(svrreply);

rc=srcstop("","",999,CANCEL,&replen,&svrreply,SSHELL);
```

   This request stops a subsystem with a stop type of CANCEL, with the process ID of 999 on the local machine and prints a message to standard output about the status of the stop.

Files

Item	Description
/etc/services	Defines sockets and protocols used for Internet services.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

srcstrt Subroutine

Purpose

Starts a System Resource Controller (SRC) subsystem.

Library

System Resource Controller Library (**libsrc.a**)

Syntax

```
#include<spc.h>
```

```
srcstrt (Host, SubsystemName, Environment, Arguments, Restart, StartFrom)
```

```
char * Host, * SubsystemName;

char * Environment, * Arguments;

unsigned int  Restart;
int  StartFrom;
```

Description

The **srcstrt** subroutine sends a start subsystem request packet and waits for a reply from the System Resource Controller (SRC).

Parameters

Item	Description
Host	Specifies the foreign host on which this start subsystem action is requested. If the host is null, the request is sent to the SRC on the local host. The local user must be running as "root". The remote system must be configured to accept remote System Resource Controller requests. That is, the **srcmstr** daemon (see **/etc/inittab**) must be started with the **-r** flag and the **/etc/hosts.equiv** or **.rhosts** file must be configured to allow remote requests.
SubsystemName	Specifies the name of the subsystem to start.
Environment	Specifies a string that is placed in the subsystem environment when the subsystem is executed. The combined values of the *Environment* and *Arguments* parameters cannot exceed a maximum of 2400 characters. Otherwise, the **srcstrt** subroutine will fail. The environment string is parsed by the SRC according to the same rules used by the shell. For example, quoted strings are passed as a single *Environment* value, and blanks outside a quoted string delimit each environment value.

Item	Description
Arguments	Specifies a string that is passed to the subsystem when the subsystem is executed. The string is parsed from the command line and appended to the command line arguments from the subsystem object class. The combined values of the *Environment* and *Arguments* parameters cannot exceed a maximum of 2400 characters. Otherwise, the **srcstrt** subroutine will fail. The command argument is parsed by the SRC according to the same rules used by the shell. For example, quoted strings are passed as a single argument, and blanks outside a quoted string delimit each argument.
Restart	Specifies override on subsystem restart. If the *Restart* parameter is set to **SRCNO**, the subsystem's restart definition from the subsystem object class is used. If the *Restart* parameter is set to **SRCYES**, the restart of a subsystem is not attempted if it terminates abnormally.
StartFrom	Specifies whether the **srcstrt** subroutine is to display start results to standard output. If the *StartFrom* parameter is set to **SSHELL**, the start results are displayed to standard output, and the **srcstrt** subroutine always returns successfully. If the *StartFrom* parameter is set to **SDAEMON**, the start results are not displayed to standard output but are passed back to the caller.

Return Values

When the *StartFrom* parameter is set to **SSHELL**, the **srcstrt** subroutine returns the value **SRC_OK**. Otherwise, it returns the subsystem process ID.

Error Codes

The **srcstrt** subroutine fails if any of the following are true:

Item	Description
SRC_AUDITID	The audit user ID is invalid.
SRC_DMNA	The SRC daemon is not active.
SRC_FEXE	The subsystem could not be **fork**ed and **exec**ed.
SRC_INET_AUTHORIZED_HOST	The local host is not in the remote **/etc/hosts.equiv** file.
SRC_INET_INVALID_HOST	On the remote host, the local host is not known.
SRC_INVALID_USER	The user is not root or group system.
SRC_INPT	The subsystem standard input file could not be established.
SRC_MMRY	An SRC component could not allocate the memory it needs.
SRC_MSGQ	The subsystem message queue could not be created.
SRC_MULT	Multiple instance of the subsystem are not allowed.
SRC_NORPLY	The request timed out waiting for a response.
SRC_OUT	The subsystem standard output file could not be established.
SRC_PIPE	A pipe could not be established for the subsystem.
SRC_SERR	The subsystem standard error file could not be established.
SRC_SUBSOCK	The subsystem communication socket could not be created.
SRC_SUBSYSID	The system user ID is invalid.
SRC_SOCK	There is a problem with SRC socket communications.
SRC_SVND	The subsystem is unknown to the SRC daemon.

Item	Description
SRC_UDP	The SRC port is not defined in the **/etc/services** header file.
SRC_UHOST	The foreign host is not known.

Examples

1. To start a subsystem passing the *Environment* and *Arguments* parameters, enter:

```
rc=srcstrt("","srctest","HOME=/tmpTERM=ibm6155",
"-z\"thezflagargument\"",SRC_YES,SSHELL);
```

   This starts the `srctest` subsystem on the local host, placing `HOME=/tmp, TERM=ibm6155` in the environment and using `-z` and `thezflagargument` as two arguments to the subsystem. This also displays the results of the start command to standard output and allows the SRC to restart the subsystem should it end abnormally.

2. To start a subsystem on a foreign host, enter:

```
rc=srcstrt("MaryC","srctest","","",SRC_NO,SDAEMON);
```

   This starts the `srctest` subsystem on the `MaryC` machine. This does not display the results of the start command to standard output and does not allow the SRC to restart the subsystem should it end abnormally.

Files

Item	Description
/etc/services	Defines sockets and protocols used for Internet services.
/dev/SRC	Specifies the **AF_UNIX** socket file.
/dev/.SRC-unix	Specifies the location for temporary socket files.

ssignal or gsignal Subroutine

Purpose

Implements a software signal facility.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <signal.h>
```

```
void (*ssignal ( Signal,  Action))( )
int Signal;
void (*Action)( );
```

```
int gsignal (Signal)
int Signal;
```

Description

Attention: Do not use the **ssignal** or **gsignal** subroutine in a multithreaded environment.

The **ssignal** and **gsignal** subroutines implement a software facility similar to that of the **signal** and **kill** subroutines. However, there is no connection between the two facilities. User programs can use the **ssignal** and **gsignal** subroutines to handle exceptional processing within an application. The **signal** subroutine and related subroutines handle system-defined exceptions.

The software signals available are associated with integers in the range 1 through 16. Other values are reserved for use by the C library and should not be used.

The **ssignal** subroutine associates the procedure specified by the *Action* parameter with the software signal specified by the *Signal* parameter. The **gsignal** subroutine raises the *Signal*, causing the procedure specified by the *Action* parameter to be taken.

The *Action* parameter is either a pointer to a user-defined subroutine, or one of the constants **SIG_DFL** (default action) and **SIG_IGN** (ignore signal). The **ssignal** subroutine returns the procedure that was previously established for that signal. If no procedure was established before, or if the signal number is illegal, then the **ssignal** subroutine returns the value of **SIG_DFL**.

The **gsignal** subroutine raises the signal specified by the *Signal* parameter by doing the following:

- If the procedure for the *Signal* parameter is **SIG_DFL**, the **gsignal** subroutine returns a value of 0 and takes no other action.
- If the procedure for the *Signal* parameter is **SIG_IGN**, the **gsignal** subroutine returns a value of 1 and takes no other action.
- If the procedure for the *Signal* parameter is a subroutine, the *Action* value is reset to the **SIG_DFL** procedure and the subroutine is called, with the *Signal* value passed as its parameter. The **gsignal** subroutine returns the value returned by the signal-handling routine.
- If the *Signal* parameter specifies an illegal value or if no procedure is specified for that signal, the **gsignal** subroutine returns a value of 0 and takes no other action.

Parameters

Item	Description
Signal	Specifies a signal.
Action	Specifies a procedure.

statacl or fstatacl Subroutine

Purpose

Retrieves the AIXC ACL type access control information for a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/acl.h>
#include <sys/stat.h>

int statacl (Path, Command, ACL, ACLSize)
char * Path;
int  Command;
```

```
struct acl * ACL;
int  ACLSize;

int fstatacl (FileDescriptor, Command, ACL, ACLSize)
int  FileDescriptor;
int Command;
struct acl *ACL;
int ACLSize;
```

Description

The **statacl** and **fstatacl** subroutines return the access control information for a file system object if the ACL associated is of AIXC type. If the ACL associated is of different type or if the underlying physical file system does not support AIXC ACL type, error could be returned by these interfaces. If the **statacl** subroutine is used on NFS V4 files, invalid results are returned.

Parameters

Item	Description
Path	Specifies a pointer to the path name of a file.
FileDescriptor	Specifies the file descriptor of an open file.
Command	Specifies the mode of the path interpretation for *Path*, specifically whether to retrieve information about a symbolic link or mount point. Valid values for the *Command* parameter are defined in the **stat.h** file and include: • **STX_LINK** • **STX_MOUNT** • **STX_NORMAL**

Item	Description
ACL	Specifies a pointer to a buffer to contain the AIXC-type Access Control List (ACL) of the file system object. The format of an AIXC ACL is defined in the **sys/acl.h** file and includes the following members: **acl_len** Size of the Access Control List (ACL). **Note:** The entire ACL for a file cannot exceed one memory page (4096 bytes). **acl_mode** File mode. **Note:** The valid values for the `acl_mode` are defined in the **sys/mode.h** file. **u_access** Access permissions for the file owner. **g_access** Access permissions for the file group. **o_access** Access permissions for default class *others*. **acl_ext[ ]** An array of the extended entries for this access control list. The members for the base ACL (owner, group, and others) may contain the following bits, which are defined in the **sys/access.h** file: **R_ACC** Allows read permission. **W_ACC** Allows write permission. **X_ACC** Allows execute or search permission.
ACLSize	Specifies the size of the buffer to contain the ACL. If this value is too small, the first word of the ACL is set to the size of the buffer needed.

Return Values

On successful completion, the **statacl** and **fstatacl** subroutines return a value of 0. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **statacl** subroutine fails if one or more of the following are true:

Item	Description
ENOTDIR	A component of the *Path* prefix is not a directory.
ENOENT	A component of the *Path* does not exist or has the **disallow truncation** attribute (see the **ulimit** subroutine).
ENOENT	The *Path* parameter was null.
EACCES	Search permission is denied on a component of the *Path* prefix.
EFAULT	The *Path* parameter points to a location outside of the allocated address space of the process.

Item	Description
ESTALE	The process' root or current directory is located in a virtual file system that has been unmounted.
ELOOP	Too many symbolic links were encountered in translating the *Path* parameter.
ENOENT	A symbolic link was named, but the file to which it refers does not exist.
ENAMETOOLONG	A component of the *Path* parameter exceeded 255 characters, or the entire *Path* parameter exceeded 1023 characters.

The **fstatacl** subroutine fails if the following is true:

Item	Description
EBADF	The file descriptor *FileDescriptor* is not valid.

The **statacl** or **fstatacl** subroutine fails if one or more of the following are true:

Item	Description
EFAULT	The *ACL* parameter points to a location outside of the allocated address space of the process.
EINVAL	The *Command* parameter is not a value of **STX_LINK**, **STX_MOUNT**, **STX_NORMAL**.
ENOSPC	The *ACLSize* parameter indicates the buffer at *ACL* is too small to hold the Access Control List. In this case, the first word of the buffer is set to the size of the buffer required.
EIO	An I/O error occurred during the operation.

If Network File System (NFS) is installed on your system, the **statacl** and **fstatacl** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

statea Subroutine

Purpose

Provides information about an extended attribute.

Syntax

```
#include <sys/ea.h>

int statea(const char *path, const char *name, struct stat64x *buffer)
int fstatea(int filedes, const char *name, struct stat64x *buffer)
int lstatea(const char *path, const char *name, struct stat64x *buffer)
```

Description

Extended attributes are name:value pairs associated with the file system objects (such as files, directories, and symlinks). They are extensions to the normal attributes that are associated with all of the objects in the file system (that is, the **stat(2)** data).

Do not define an extended attribute name with the 8-character prefix "(0xF8)SYSTEM(0xF8)". Prefix "(0xF8)SYSTEM(0xF8)" is reserved for system use only.

Note: 0xF8 represents a non-printable character.

The **statea** subroutine gets information about the extended attribute name *name* associated with the file system object specified by *path*. The **fstatea** subroutine is identical to **statea**, except that it takes a file descriptor instead of a path. The **lstatea** subroutine is identical to **statea**, except, in the case of a symbolic link, the link itself is interrogated rather than the file that it refers to.

The **statea** subroutine uses a **stat64x** structure to return the information. Note that all values in this structure are 64-bit, including the devices and size. A normal **struct stat** cannot be passed to **statea**. For more information, see the "stat, fstat, lstat, statx, fstatx, statxat, fstatat, fullstat, ffullstat, stat64, fstat64, lstat64, stat64x, fstat64x, lstat64x, or stat64xat Subroutine" on page 2062.

Parameters

Item	Description
path	The path name of the file.
name	The name of the extended attribute. An extended attribute name is a NULL-terminated string.
buffer	A pointer to the stat structure in which information is returned.
filedes	A file descriptor for the file.

Return Values

If the **statea** subroutine succeeds, 0 is returned. Upon failure, -1 is returned and **errno** is set appropriately.

Error Codes

Item	Description
EACCES	Caller lacks read permission on the base file, or lacks the appropriate ACL privileges for named attribute **lookup**.
EFAULT	A bad address was passed for *path*, *name*, or *buffer*.
EFORMAT	File system is capable of supporting EAs, but EAs are disabled.
EINVAL	A path-like name should not be used (such as **zml/file**, **.** and **..**).
ENAMETOOLONG	The *path* or *name* value is too long.
ENOATTR	No attribute named *name* is present.
ENOTSUP	Extended attributes are not supported by the file system.

standend, standout, wstandend, or wstandout Subroutine

Purpose

Sets and clears window attributes.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int standend
(void);

int standout
(void);

int wstandend
(WINDOW *win);

int wstandout
(WINDOW  *win);
```

Description

The **standend** and **standout** subroutines turn off all attributes of the current or specified window.

The **wstandout** and **wstandend** subroutines turn on the **standout** attribute of the current or specified window.

Parameters

Item	Description
**win*	Specifies the window in which to set the attributes.

Return Values

These subroutines always return 1.

Examples

1. To turn on the **standout** attribute in the stdscr, enter:

   ```
   standout();
   ```

   This example is functionally equivalent to:

   ```
   attron(A_STANDOUT);
   ```

2. To turn on the **standout** attribute in the user-defined window `my_window`, enter:

   ```
   WINDOW *my_window;
   wstandout(my_window);
   ```

   This example is functionally equivalent to:

   ```
   wattron(my_window, A_STANDOUT);
   ```

3. To turn off the **standout** attribute in the default window, enter:

   ```
   standend();
   ```

   This example is functionally equivalent to:

   ```
   attroff(A_STANDOUT);
   ```

4. To turn off the **standout** attribute in the user-defined window `my_window`, enter:

   ```
   WINDOW *my_window;
   wstandend(my_window);
   ```

   This example is functionally equivalent to:

```
wattroff(my_window, A_STANDOUT);
```

start_color Subroutine

Purpose

Initializes color.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
start_color()
```

Description

The **start_color** subroutine initializes color. This subroutine requires no arguments. You must call the **start_color** subroutine if you intend to use color in your application. Except for the **has_colors** and **can_change_color** subroutines, you must call the **start_color** subroutine before any other color manipulation subroutine. A good time to call **start_color** is right after calling the **initscr** routine and after establishing whether the terminal supports color.

The **start_color** routine initializes the following basic colors:

Item	Description
COLOR_BLACK	0
COLOR_BLUE	1
COLOR_GREEN	2
COLOR_CYAN	3
COLOR_RED	4
COLOR_MAGENTA	5
COLOR_YELLOW	6
COLOR_WHITE	7

The subroutine also initializes two global variables: **COLORS** and **COLOR_PAIRS**. The **COLORS** variable is the maximum number of colors supported by the terminal. The **COLOR_PAIRS** variable is the maximum number of color-pairs supported by the terminal.

The **start_color** subroutine also restores the terminal's colors to the original values right after the terminal was turned on.

Return Values

Item	Description
ERR	Indicates the terminal does not support colors.

Item	Description
OK	Indicates the terminal does support colors.

Example

To enable the color support for a terminal that supports color, use:

```
start_color();
```

statfs, fstatfs, statfs64, fstatfs64, or ustat Subroutine

Purpose

Gets file system statistics.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/statfs.h>
```

int statfs (*Path*, *StatusBuffer*)
char *Path;
struct statfs *StatusBuffer;

int fstatfs (*FileDescriptor*, *StatusBuffer*)
int *FileDescriptor*;
struct statfs *StatusBuffer;

int statfs64 (*Path*, *StatusBuffer64*)
char *Path;
struct statfs64 *StatusBuffer64;

int fstatfs64 (*FileDescriptor*, *StatusBuffer64*)
int *FileDescriptor*;
struct statfs64 *StatusBuffer64;

```
#include <sys/types.h>
#include <ustat.h>
```

int ustat (*Device*, *Buffer*)
dev_t *Device*;
struct ustat *Buffer;

Description

The **statfs** and **fstatfs** subroutines return information about the mounted file system that contains the file named by the *Path* or *FileDescriptor* parameters. The returned information is in the format of a **statfs** structure, described in the **sys/statfs.h** file.

The `statfs64` and `fstatfs64` subroutines are similar to the `statfs` and `fstatfs` subroutines except that the returned information is in the format of a `statfs64` structure, described in the `sys/statfs.h` file, instead of a `statfs` structure.

The `statfs64` structure provides invariant 64-bit fields for the file system blocks (or inodes) sizes or counts, and the file system ID. This structure allows `statfs64` and `fstatfs64` to always return the specified information in invariant 64-bit sizes.

The **ustat** subroutine also returns information about a mounted file system identified by *Device*. This device identifier is for any given file and can be determined by examining the `st_dev` field of the **stat** structure defined in the **sys/stat.h** file. The returned information is in the format of a **ustat** structure**,** described in the **ustat.h** file. The **ustat** subroutine is superseded by the **statfs** and **fstatfs** subroutines. Use one of these (**statfs** and **fstatfs**) subroutines instead.

Note: The **ustat** subroutine does not work for 64-bit sizes.

Parameters

Item	Description
Path	The path name of any file within the mounted file system.
FileDescriptor	A file descriptor obtained by a successful **open** or **fcntl** subroutine. A file descriptor is a small positive integer used instead of a file name.
StatusBuffer	A pointer to a **statfs** buffer for the returned information from the **statfs** or **fstatfs** subroutine.
StatusBuffer64	A pointer to a **statfs64** buffer for the returned information from the **statfs64** or **fstatfs64** subroutine.
Device	The ID of the device. It corresponds to the `st_rdev` field of the structure returned by the **stat** subroutine. The **stat** subroutine and the **sys/stat.h** file provide more information about the device driver.
Buffer	A pointer to a **ustat** buffer to hold the returned information.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **statfs**, **fstatfs**, `statfs64`, `fstatfs64`, and **ustat** subroutines fail if the following is true:

Item	Description
EFAULT	The *Buffer* parameter points to a location outside of the allocated address space of the process.

The **fstatfs** or `fstatfs64` subroutine fails if the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid file descriptor.
EIO	An I/O error occurred while reading from the file system.

The **statfs** or `statfs64` subroutine can be unsuccessful for other reasons.

statvfs, fstatvfs, statvfs64, or fstatvfs64 Subroutine

Purpose

Returns information about a file system.

Library

Standard C Library (**libc.a)**

Syntax

```
#include <sys/statvfs.h>
```

```
int statvfs ( Path,  Buf)
const char *Path;
struct statvfs *Buf;

int fstatvfs ( Fildes, Buf)
int Fildes;
struct statvfs *Buf;

int statvfs64 ( Path,  Buf)
const char *Path;
struct statvfs64 *Buf;

int fstatvfs64 ( Fildes, Buf)
int Fildes;
struct statvfs64 *Buf;
```

Description

The **statvfs** and **fstatvfs** subroutines return descriptive information about a mounted file system containing the file referenced by the *Path* or *Fildes* parameters. The *Buf* parameter is a pointer to a structure which will by filled by the subroutine call.

The *Path* and *Fildes* parameters must reference a file which resides on the file system. Read, write, or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

The `statvfs64` and `fstatvfs64` subroutines are similar to the `statvfs` and `fstatvfs` subroutines except that the returned information is in the format of a `statvfs64` structure instead of a `statvfs` structure.

The `statvfs64` structure provides invariant 64-bit fields for the file system blocks (or inodes) sizes and counts, and the file system ID. This structure allows `statvfs64` and `fstatvfs64` to always return the specified information in invariant 64-bit values.

Parameters

Item	Description
Path	The path name identifying the file.
Buf	A pointer to a **statvfs** or `statvfs64` structure in which information is returned. The **statvfs** or `statvfs64` structure is described in the **sys/statvfs.h** header file.
Fildes	The file descriptor identifying the open file.

Return Values

Item	Description
0	Successful completion.
-1	Not successful and errno set to one of the following.

Error Codes

Item	Description
EACCES	Search permission is denied on a component of the path.
EBADF	The file referred to by the *Fildes* parameter is not an open file descriptor.
EIO	An I/O error occurred while reading from the filesystem.
ELOOP	Too many symbolic links encountered in translating path.
ENAMETOOLONG	The length of the pathname exceeds **PATH_MAX**, or name component is longer than **NAME_MAX.**
ENOENT	The file referred to by the *Path* parameter does not exist.
ENOMEM	A memory allocation failed during information retrieval.
ENOTDIR	A component of the *Path* parameter prefix is not a directory.
EOVERFLOW	One of the values to be returned cannot be represented correctly in the structure pointed to by **buf**.

stat, fstat, lstat, statx, fstatx, statxat, fstatat, fullstat, ffullstat, stat64, fstat64, lstat64, stat64x, fstat64x, lstat64x, or stat64xat Subroutine

Purpose

Provides information about a file or shared memory object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/stat.h>
```

```
int stat (Path, Buffer)
const char *Path;
struct stat *Buffer;
```

```
int fstatat (DirFileDescriptor, Path, Buffer, Flag)
int DirFileDescriptor;
const char * Path;
struct stat * Buffer;
int Flag;
```

```
int lstat (Path,Buffer)
const char *Path;
struct stat *Buffer;
```

```
int fstat (FileDescriptor, Buffer)
int FileDescriptor;
struct stat *Buffer;
```

```
int statx (Path, Buffer, Length, Command)
char *Path;
struct stat *Buffer;
int Length;
int Command;
```

```
int statxat (DirFileDescriptor, Path, Buffer, Length, Command)
int DirFileDescriptor;
char * Path;
struct stat *Buffer;
int Length;
int Command;
```

```
int fstatx (FileDescriptor, Buffer, Length, Command)
int FileDescriptor;
struct stat *Buffer;
int Length;
int Command;
```

```
int stat64 (Path, Buffer)
const char *Path;
struct stat64 *Buffer;
```

```
int stat64at (DirFileDescriptorPath, BufferFlag)
int DirFileDescriptor
const char *Path;
struct stat64 *Buffer;
int Flag;
```

```
int lstat64 (Path, Buffer)
const char *Path;
struct stat64 *Buffer;
```

```
int fstat64 (FileDescriptor, Buffer)
int FileDescriptor;
struct stat64 *Buffer;
```

```
int stat64x (Path,Buffer)
const char *Path;
struct stat64x *Buffer;
```

```
int stat64xat(DirFileDescriptor, Path, Buffer, Flag)
int DirFileDescriptor;
const char * Path;
struct stat64x * Buffer;
int Flag;
```

```
int lstat64x (Path,Buffer)
const char *Path;
struct stat64x *Buffer;
```

```
int fstat64x (FileDescriptor,Buffer)
int FileDescriptor;
struct stat64x *Buffer;
```

```
#include <sys/fullstat.h>
```

```
int fullstat (Path,Command, Buffer)
struct fullstat *Buffer;
char *Path;
int Command;
```

```
int ffullstat (FileDescriptor,Command, Buffer)
int FileDescriptor;
int Command;
struct fullstat *Buffer;
```

Description

The **stat** subroutine obtains information about the file named by the *Path* parameter. Read, write, or execute permission for the named file is not required, but all directories listed in the path leading to the file must be searchable. The file information, which is a subset of the **stat** structure, is written to the area specified by the *Buffer* parameter.

The **lstat** subroutine obtains information about a file that is a symbolic link. The **lstat** subroutine returns information about the link, while the **stat** subroutine returns information about the file referenced by the link.

The **fstat** subroutine obtains information about the open file or shared memory object referenced by the *FileDescriptor* parameter. The **fstatx** subroutine obtains information about the open file or shared memory object referenced by the *FileDescriptor* parameter, as in the **fstat** subroutine.

The `st_mode`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime`, and `st_mtime` fields of the **stat** structure have meaningful values for all file types. The **statx**, **stat**, **lstat**, **fstatx**, **fstat**, **fullstat**, or **ffullstat** subroutine sets the `st_nlink` field to a value equal to the number of links to the file.

The **statx** subroutine obtains a greater set of file information than the **stat** subroutine. The *Path* parameter is processed differently, depending on the contents of the *Command* parameter. The *Command* parameter provides the ability to collect information about symbolic links (as with the **lstat** subroutine) as well as information about mount points and hidden directories. The **statx** subroutine returns the amount of information specified by the *Length* parameter.

The **fullstat** and **ffullstat** subroutines are interfaces maintained for backward compatibility. With the exception of some field names, the **fullstat** structure is identical to the **stat** structure.

The **stat64**, **lstat64**, and **fstat64** subroutines are similar to the **stat**, **lstat**, **fstat** subroutines except that they return file information in a **stat64** structure instead of a **stat** structure. The information is identical except that the `st_size` field is defined to be a 64-bit size. This allows **stat64**, **lstat64**, and **fstat64** to return file sizes which are greater than **OFF_MAX** (2 gigbytes minus 1).

In the large file enabled programming environment, **stat** is redefined to be **stat64**, **lstat** is redefined to be **lstat64** and **fstat** is redefined to be **fstat64**.

The **stat64x**, **lstat64x**, and **fstat64x** subroutines are similar to the **stat**, **lstat**, **fstat** subroutines except that they return file information in a **stat64x** structure instead of a **stat** structure. The information is identical except the following fields are defined to be 64-bit sizes: **st_dev**, **st_ino**, **st_rdev**, **st_size**, **st_atime**, **st_mtime**, **st_ctime**, **st_blksize**, and **st_blocks**.

Note: The 64-bit **st_dev** field always contains a 64-bit device ID, where the first two bits are reserved, the next 30 bits are the device major number, and the next 32 bits are the device minor number.

This allows **stat64x**,**fstat64x**, and **lstat64x** to return the specified information in invariant 64-bit sizes, regardless of the mode of an application or the kernel it is running on.

If the i-node number is larger than the maximum number that can be represented in the **stat** structure, the returned i-node number has a value of -1. In this condition, use the **stat64x** subroutine to retrieve the accurate i-node number.

The **statxat** subroutine is equivalent to the **statx** subroutine if the *DirFileDescriptor* parameter is **AT_FDCWD** or the *Path* parameter is an absolute path name. If *DirFileDescriptor* is a valid file descriptor of an open directory and *Path* is a relative path name, *Path* is considered to be relative to the directory associated with the *DirFileDescriptor* parameter instead of the current working directory.

Similarly, the **fstatat**, **stat64at**, or **stat64xat** subroutine is equivalent to the **stat**, **stat64**, or **stat64x** subroutine, respectively, in the same way as **statx** and **statxat** if the *Flag* parameter does not have the **AT_SYMLINK_NOFOLLOW** bit set.

If the *Flag* parameter does have the **AT_SYMLINK_NOFOLLOW** bit set in the **fstatat**, **stat64at**, or **stat64xat** subroutine, then it is equivalent to the **lstat**, **lstat64**, or **lstat64x** subroutine, respectively.

Parameters

DirFileDescriptor
Specifies the file descriptor of an open directory.

Path
Specifies the path name identifying the file. This name is interpreted differently depending on the interface used. If *DirFileDescriptor* is specified and *Path* is a relative path name, then *Path* is considered relative to the directory specified by *DirFileDescriptor*.

Flag
Specifies a bit field. If it contains the **AT_SYMLINK_NOFOLLOW** bit and *Path* points to a symbolic link, the information for the symbolic link is returned.

FileDescriptor
Specifies the file descriptor identifying the open file or shared memory object.

Note: If the *FileDescriptor* parameter references a shared memory object, only the *st_uid*, *st_gid*, *st_size*, and *st_mode* fields of the **stat** structure are filled, and only the **S_IRUSR**, **S_IWUSR**, **S_IRGRP**, **S_IWGRP**, **S_IROTH**, and **S_IWOTH** file permission bits are valid.

Buffer
Specifies a pointer to the **stat** structure in which information is returned. The **stat** structure is described in the **<sys/stat.h>** file.

Length
Indicates the amount of information, in bytes, to be returned. Any value between 0 and the value returned by the **STATXSIZE** macro, inclusive, may be specified. The following macros may be used:

STATSIZE
Specifies the subset of the **stat** structure that is normally returned for a **stat** call.

FULLSTATSIZE
Specifies the subset of the **stat** (**fullstat**) structure that is normally returned for a **fullstat** call.

STATXSIZE
Specifies the complete **stat** structure. 0 specifies the complete **stat** structure, as if **STATXSIZE** had been specified.

Command
Specifies a processing option. For the **statx** subroutine, the *Command* parameter determines how to interpret the path name provided, specifically, whether to retrieve information about a symbolic link, hidden directory, or mount point. Flags can be combined by logically ORing them together. The following options are possible values:

STX_LINK
If the *Command* parameter specifies the **STX_LINK** flag and the *Path* parameter is a path name that refers to a symbolic link, the **statx** subroutine returns information about the symbolic link. If the **STX_LINK** flag is not specified, the **statx** subroutine returns information about the file to which the link refers.

If the *Command* parameter specifies the **STX_LINK** flag and the *Path* value refers to a symbolic link, the `st_mode` field of the returned **stat** structure indicates that the file is a symbolic link.

STX_HIDDEN
If the *Command* parameter specifies the **STX_HIDDEN** flag and the *Path* value is a path name that refers to a hidden directory, the **statx** subroutine returns information about the hidden directory. If the **STX_HIDDEN** flag is not specified, the **statx** subroutine returns information about a subdirectory of the hidden directory.

If the *Command* parameter specifies the **STX_HIDDEN** flag and *Path* refers to a hidden directory, the `st_mode` field of the returned **stat** structure indicates that this is a hidden directory.

STX_MOUNT
If the *Command* parameter specifies the **STX_MOUNT** flag and the *Path* value is the name of a file or directory that has been mounted over, the **statx** subroutine returns information about the mounted-over file. If the **STX_MOUNT** flag is not specified, the **statx** subroutine returns information about the mounted file or directory (the root directory of a virtual file system).

If the *Command* parameter specifies the **STX_MOUNT** flag, the **FS_MOUNT** bit in the `st_flag` field of the returned **stat** structure is set if, and only if, this file is mounted over.

If the *Command* parameter does not specify the **STX_MOUNT** flag, the **FS_MOUNT** bit in the `st_flag` field of the returned **stat** structure is set if, and only if, this file is the root directory of a virtual file system.

STX_NORMAL
If the *Command* parameter specifies the **STX_NORMAL** flag, then no special processing is performed on the *Path* value. This option should be used when **STX_LINK**, **STX_HIDDEN**, and **STX_MOUNT** flags are not desired.

For the **fstatx** subroutine, there are currently no special processing options. The only valid value for the *Command* parameter is the **STX_NORMAL** flag.

For the **fullstat** and **ffullstat** subroutines, the *Command* parameter may specify the **FL_STAT** flag, which is equivalent to the **STX_NORMAL** flag, or the **FL_NOFOLLOW** flag, which is equivalent to **STX_LINK** flag.

STX_64
If the *Command* parameter specifies the **STX_64** flag and the file size is greater than **OFF_MAX**, then **statx** succeeds and returns the file size. Otherwise, **statx** fails and sets the **errno** to **EOVERFLOW**.

STX_64X
If the *Command* parameter specifies the **STX_64X** flag and the **stat** structure size is not equal to the size of **STX_64X**, **statx** fails and sets the **errno** to **EINVAL**.

STX_EFSRAW
If the *Command* parameter specifies the **STX_EFSRAW** flag and the *Path* parameter is a path name that refers to an encrypted file, the **statx** subroutine returns the full encrypted size of the file.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **stat**, **fstatat**, **lstat**, **statx**, **statxat**, and **fullstat** subroutines are unsuccessful if one or more of the following are true:

Item	Description
EACCES	Search permission is denied for one component of the path prefix.
ENAMETOOLONG	The length of the path prefix exceeds the **PATH_MAX** flag value or a path name is longer than the **NAME_MAX** flag value while the **POSIX_NO_TRUNC** flag is in effect.
ENOTDIR	A component of the path prefix is not a directory.
EFAULT	Either the *Path* or the *Buffer* parameter points to a location outside of the allocated address space of the process.
ENOENT	The file named by the *Path* parameter does not exist.

Item	Description
EOVERFLOW	The file size is larger than the maximum value that can be represented in the **stat** structure pointed to by the *Buffer* parameter.

The **stat**, **fstatat**,**lstat**, **statx**, **statxat**, and **fullstat** subroutines can be unsuccessful for other reasons.

The **fstat**,**fstatx**, and**ffullstat** subroutines fail if one or more of the following are true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid file descriptor.
EFAULT	The *Buffer* parameter points to a location outside the allocated address space of the process.
EIO	An input/output (I/O) error occurred while reading from the file system.

The **statx**, **statxat**, and **fstatx** subroutines are unsuccessful if one or more of the following are true:

Item	Description
EINVAL	The *Length* value is not between 0 and the value returned by the **STATSIZE** macro, inclusive.
EINVAL	The *Command* parameter contains an unacceptable value.

The **statxat**, **fstatat**, **stat64at**, and **stat64xat** subroutines are unsuccessful if one or more of the following are true:

Item	Description
EBADF	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a valid file descriptor.
ENOTDIR	The Path parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.

The **fstatat**, **stat64at**, and **stat64xat** subroutines are unsuccessful if the following is true:

Item	Description
EINVAL	The *Flag* parameter is invalid.

Files

Item	Description
/usr/include/sys/fullstat.h	Contains the **fullstat** structure.
/usr/include/sys/mode.h	Defines values on behalf of the **stat.h** file.

strcat, strncat, strxfrm, strxfrm_l, strcpy, strncpy, stpcpy, stpncpy, strdup or strndup Subroutines

Purpose

Copies and appends strings in memory.

Library

Standard C Library (**libc.a**)

Syntax

#include <string.h>

char * strcat (*String1*, *String2***) char ***String1***; const char ***String2**;

char * strncat (*String1***,** *String2*, *Number***) char ***String1***; const char ***String2***; size_t** *Number***;**

size_t strxfrm (*String1***,** *String2*, *Number***) char ***String1***; const char ***String2***; size_t** *Number***;**

size_t strxfrm_l (*String1***,** *String2, Number*,*Locale***) char ***String1***; const char ***String2***; size_t** *Number***; locale_t** *Locale***;**

char * strcpy (*String1***,** *String2***) char ***String1***; const char ***String2**;

char * strncpy (*String1***,** *String2*, *Number***) char ***String1***; const char ***String2***; size_t** *Number***;**

char * stpcpy (*String1***,** *String2***) char ***String1***; const char ***String2**;

char * stpncpy (*String1***,** *String2*, *size***) char ***String1***; const char ***String2***; size_t** *size***;**

char * strdup (*String1***) const char ***String1**;

char * strndup (*String1***,** *size***) const char ***String1***; size_t** *size***;**

Description

The **strcat**, **strncat**, **strxfrm**, **strcpy**, **strxfrm_l**, **strncpy**, **stpcpy**, **stpncpy**, **strdup**, and **strndup** subroutines copy and append strings in memory.

The *String1* and *String2* parameters point to strings. A string is an array of characters terminated by a null character. The **strcat**, **strncat**, **strcpy**, and **strncpy** subroutines all alter the string in the *String1* parameter. However, they do not check for overflow of the array to which the *String1* parameter points. String movement is performed on a character-by-character basis and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** file.

The **strcat** subroutine adds a copy of the string pointed to by the *String2* parameter to the end of the string pointed to by the *String1* parameter. The **strcat** subroutine returns a pointer to the null-terminated result.

The **strncat** subroutine copies a number of bytes specified by the *Number* parameter from the *String2* parameter to the end of the string pointed to by the *String1* parameter. The subroutine stops copying before the end of the number of bytes specified by the *Number* parameter if it encounters a null character in the *String2* parameter's string. The **strncat** subroutine returns a pointer to the null-terminated result. The **strncat** subroutine returns the value of the *String1* parameter.

The **strxfrm** subroutine transforms the string pointed to by the *String2* parameter and places it in the array pointed to by the *String1* parameter. The **strxfrm** subroutine transforms the entire string if possible, but places no more than the number of bytes specified by the *Number* parameter in the array pointed to by the *String1* parameter. Consequently, if the *Number* parameter has a value of 0, the *String1* parameter can be a null pointer. The **strxfrm** subroutine returns the length of the transformed string, not including the terminating null byte. If the returned value is equal to or more than that of the *Number* parameter, the contents of the array pointed to by the *String1* parameter are indeterminable. If the number of bytes specified by the *Number* parameter is 0, the **strxfrm** subroutine returns the length required to store the transformed string, not including the terminating null byte. The **strxfrm** subroutine is determined by the **LC_COLLATE** category.

The *strxfrm_l()* function is equivalent to the *strxfrm()* function, except that the locale data used is from the locale represented by *Locale*.

The **strcpy** and **stpcpy** subroutines copy the string pointed to by the *String2* parameter to the character array pointed to by the *String1* parameter. Copying stops after the null character is copied. The **strcpy** subroutine returns the value of the *String1* parameter, if successful. Otherwise, a null pointer is returned.

The **stpcpy** subroutines returns a pointer to the terminating NULL character copied into the *String1* parameter, if successful. Otherwise, a null pointer is returned.

The **strncpy** and **stpncpy** subroutines copy the number of bytes specified by the *Number* parameter from the string pointed to by the *String2* parameter to the character array pointed to by the *String1* parameter. If the *String2* parameter value is less than the specified number of characters, then the **strncpy** subroutine pads the *String1* parameter with trailing null characters to a number of bytes equaling the value of the *Number* parameter. If the *String2* parameter is exactly the specified number of characters or more, then only the number of characters specified by the *Number* parameter are copied and the result is not terminated with a null byte. The **strncpy** subroutine returns the value of the *String1* parameter.

If a null character is written to the destination, the **stpncpy** function returns the address of the first such null character. Otherwise, it returns *&String1***[***Number***]**.

The **strdup** subroutine returns a pointer to a new string, which is a duplicate of the string pointed to by the *String1* parameter. Space for the new string is obtained by using the **malloc** subroutine. A null pointer is returned if the new string cannot be created.

The **strndup** subroutine is equivalent to the **strdup** subroutine, except that it copies at most *size* plus one byte into the newly allocated memory, terminating the new string with a null character. If the length of *String1* is larger than *size*, only *size* bytes is duplicated. If *size* is larger than the length of *String1*, all bytes in *String1* shall be copied into the new memory buffer, including the terminating NULL character

Parameters

Item	Description
Number	Specifies the number of bytes in a string to be copied or transformed.
String1	Points to a string to which the specified data is copied or appended.
String2	Points to a string which contains the data to be copied, appended, or transformed.
Locale	Specifies the locale in which the string has to be transformed.

Error Codes

The **strcat**, **strncat**, **strxfrm**, **strxfrm_l**, **strcpy**, **strncpy**, **stpcpy**, **stpncpy**,**strdup**, and **strndup** subroutines fail if the following occurs:

Item	Description
EFAULT	A string parameter is an invalid address.

In addition, the **strxfrm**, and **strxfrm_l** subroutine fails if:

Item	Description
EINVAL	A string parameter contains characters outside the domain of the collating sequence.

The **strdup** and **strndup** functions fails if:

Item	Description
ENOMEM	Storage space available is insufficient.

strcmp, strncmp, strcasecmp, strcasecmp_l , strncasecmp, strncasecmp_l, strcoll, or strcoll_l Subroutine

Purpose

Compares strings in memory.

Library

Standard C Library (**libc.a**)

Syntax

#include <string.h>

int strcmp (*String1*, *String2***) const char ***String1**, ***String2**;**

int strncmp (*String1***,** *String2*, *Number***) const char ***String1**, ***String2**; size_t** *Number***;**

int strcoll (*String1***,** *String2***) const char ***String1**, ***String2**;**

int strcoll_l (*String1***,** *String2*,*Locale***) const char ***String1**, ***String2**;locale_t** *Locale;*

#include <strings.h>

int strcasecmp (*String1***,** *String2***) const char ***String1**, ***String2**;**

int strcasecmp_l (*String1***,** *String2*,*Locale***) const char ***String1**, ***String2**;locale_t** *Locale;*

int strncasecmp (*String1***,** *String2***,** *Number***) const char ***String1**, ***String2**; size_t** *Number***;**

int strncasecmp_l (*String1***,** *String2*,*Number*,*Locale***)**
const char ***String1, ***String2**;size_t** *Number***;locale_t** *Locale;*

Description

The **strcmp**, **strncmp**, **strcasecmp**, **strcasecmp_l**, **strncasecmp**, **strncasecmp_l**, **strcoll**, and **strcoll_l** subroutines compare strings in memory.

The *strcasecmp_l()*, *strncasecmp_l()*, and *strcol_ll()* functions are the same as *strcasecmp()*, *strncasecmp()*, and *strcoll()* functions except that they use the locale represented by *Locale* to determine the case of the characters instead of the current locale.

The *String1* and *String2* parameters point to strings. A string is an array of characters terminated by a null character.

The **strcmp** subroutine performs a case-sensitive comparison of the string pointed to by the *String1* parameter and the string pointed to by the *String2* parameter, and analyzes the extended ASCII character set values of the characters in each string. The **strcmp** subroutine compares **unsigned char** data types. The **strcmp** subroutine then returns a value that is:

- Less than 0 if the value of string *String1* is lexicographically less than string *String2*.
- Equal to 0 if the value of string *String1* is lexicographically equal to string *String2*.
- Greater than 0 if the value of string *String1* is lexicographically greater than string *String2*.

The **strncmp** subroutine makes the same comparison as the **strcmp** subroutine, but compares up to the maximum number of pairs of bytes specified by the *Number* parameter.

The **strcasecmp** subroutine performs a character-by-character comparison similar to the **strcmp** subroutine. However, the **strcasecmp** subroutine is not case-sensitive. Uppercase and lowercase letters are mapped to the same character set value. The sum of the mapped character set values of each string is used to return a value that is:

- Less than 0 if the value of string *String1* is lexicographically less than string *String2*.

- Equal to 0 if the value of string *String1* is lexicographically equal to string *String2*.
- Greater than 0 if the value of string *String1* is lexicographically greater than string *String2*.

The **strncasecmp** subroutine makes the same comparison as the **strcasecmp** subroutine, but compares up to the maximum number of pairs of bytes specified by the *Number* parameter.

Note: Both the **strcasecmp** and **strncasecmp** subroutines only work with 7-bit ASCII characters.

The **strcoll** subroutine works the same as the **strcmp** subroutine, except that the comparison is based on a collating sequence determined by the **LC_COLLATE** category. If the **strcmp** subroutine is used on transformed strings, it returns the same result as the **strcoll** subroutine for the corresponding untransformed strings.

Parameters

Item	Description
Number	The number of bytes in a string to be examined.
String1	Points to a string which is compared.
String2	Points to a string which serves as the source for comparison.
Locale	Points to the locale in which the strings are compared.

Error Codes

The **strcmp**, **strncmp**, **strcasecmp**, **strncasecmp**, **strcoll**, **strcasecmp_l**, **strncasecmp_l**, and **strcoll_l** subroutines fail if the following occurs:

Item	Description
EFAULT	A string parameter is an invalid address.

In addition, the **strcoll**, and **strcoll_l** subroutines fails if:

Item	Description
EINVAL	A string parameter contains characters outside the domain of the collating sequence.

strerror Subroutine

Purpose

Maps an error number to an error message string.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>

char *strerror ( ErrorNumber)
int ErrorNumber;
```

Description

Attention: Do not use the **strerror** subroutine in a multithreaded environment.

The **strerror** subroutine maps the error number in the *ErrorNumber* parameter to the error message string. The **strerror** subroutine retrieves an error message based on the current value of the **LC_MESSAGES** category. If the specified message catalog cannot be opened, the default message is returned. The returned message does not contain a new line ("\n").

Parameters

Item	Description
ErrorNumber	Specifies the error number to be associated with the error message.

Return Values

The **strerror** subroutine returns a pointer to the error message.

strfmon, or strfmon_l Subroutine

Purpose

Formats monetary strings.

Library

Standard C Library (**libc. a**)

Syntax

```
#include <monetary.h>

ssize_t strfmon ( S,  MaxSize,  Format, ...)
char *S;
size_t MaxSize;
const char *Format, ...;

ssize_t strfmon_l ( S,  MaxSize,  Locale,  Format, ...)
char *S;
size_t MaxSize;
locale_t Locale;
const char *Format, ...;
```

Description

The **strfmon** subroutine converts numeric values to monetary strings according to the specifications in the *Format* parameter. This parameter also contains numeric values to be converted. Characters are placed into the *S* array, as controlled by the *Format* parameter. The **LC_MONETARY** category governs the format of the conversion.

The **strfmon** subroutine can be called multiple times by including additional **format** structures, as specified by the *Format* parameter.

The *Format* parameter specifies a character string that can contain plain characters and conversion specifications. Plain characters are copied to the output stream. Conversion specifications result in the fetching of zero or more arguments, which are converted and formatted.

If there are insufficient arguments for the *Format* parameter, the results are undefined. If arguments remain after the *Format* parameter is exhausted, the excess arguments are ignored.

A conversion specification consists of the following items in the following order: a % (percent sign), optional flags, optional field width, optional left precision, optional right precision, and a required conversion character that determines the conversion to be performed.

The *strfmon_l()* function is equivalent to the *strfmon()* function, except that the locale data used is from the locale represented by *Locale*.

Parameters

Item	Description
S	Contains the output of the **strfmon** subroutine.
MaxSize	Specifies the maximum number of bytes (including the null terminating byte) that may be placed in the *S* parameter.
Format	Contains characters and conversion specifications.

Flags

One or more of the following flags can be specified to control the conversion:

Item	Description
=*f*	An **=** (equal sign) followed by a single character that specifies the numeric fill character. The default numeric fill character is the space character. This flag does not affect field-width filling, which always uses the space character. This flag is ignored unless a left precision is specified.
^	Does not use grouping characters when formatting the currency amount. The default is to insert grouping characters if defined for the current locale.
+ or **(**	Determines the representation of positive and negative currency amounts. Only one of these flags may be specified. The locale's equivalent of **+** (plus sign) and **-** (negative sign) are used if **+** is specified. The locale's equivalent of enclosing negative amounts within parentheses is used if **(** (left parenthesis) is specified. If neither flag is included, a default specified by the current locale is used.
-	Left-justifies all fields (pads to the right). The default is right-justification.
!	Suppresses the currency symbol from the output conversion.

Field Width

Item	Description
w	The decimal-digit string *w* specifies the minimum field width in which the result of the conversion is right-justified. If **-***w* is specified, the result is left-justified. The default is a value of 0.

Left Precision

Item	Description
#*n*	A # (pound sign) followed by a decimal-digit string, *n*, specifies the maximum number of digits to be formatted to the left of the radix character. This option can be specified to keep formatted output from multiple calls to the **strfmon** subroutine aligned in the same columns. It can also be used to fill unused positions with a special character (for example, $***123.45). This option causes an amount to be formatted as if it has the number of digits specified by the *n* variable. If more than *n* digit positions are required, this option is ignored. Digit positions in excess of those required are filled with the numeric fill character set with the **=***f* flag. If defined for the current locale and not suppressed with the **^** flag, the subroutine inserts grouping characters before fill characters (if any). Grouping characters are not applied to fill characters, even if the fill character is a digit. In the example: `$0000001,234.56` grouping characters do not appear after the first or fourth 0 from the left. To ensure alignment, any characters appearing before or after the number in the formatted output, such as currency or sign symbols, are padded as necessary with space characters to make their positive and negative formats equal in length.

Right Precision

Item	Description
.*p*	A . (period) followed by a decimal digit string, *p*, specifies the number of digits after the radix character. If the value of the *p* variable is 0, no radix character is used. If a right precision is not specified, a default specified by the current locale is use. The amount being formatted is rounded to the specified number of digits prior to formatting.

Conversion Characters

Item	Description
i	The double argument is formatted according to the current locale's international currency format; for example, in the U.S.: 1,234.56.
n	The double argument is formatted according to the current locale's national currency format; for example, in the U.S.: $1,234.56.
%	No argument is converted; the conversion specification **%%** is replaced by a single **%**.

Return Values

If successful, and if the number of resulting bytes (including the terminating null character) is not more than the number of bytes specified by the *MaxSize* parameter, the **strfmon**, and **strfmon_l** subroutines return the number of bytes placed into the array pointed to by the *S* parameter (not including the terminating null byte). Otherwise, a value of -1 is returned and the contents of the *S* array are indeterminate.

Error Codes

The **strfmon**, and **strfmon_l** subroutines may fail if the following is true:

Item	Description
E2BIG	Conversion stopped due to lack of space in the buffer.

strftime or strftime_l Subroutine

Purpose

Formats time and date.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>
size_t strftime ( String, Length, Format, TmDate)
char *String;
size_t Length;
const char *Format;
const struct tm *TmDate;
```

```
size_t strftime_l(char *restrict String, size_t Length,
const char *restrict Format, const struct tm *restrict TmDate,locale_t Locale);
```

Description

The **strftime** subroutine converts the internal time and date specification of the **tm** structure, which is pointed to by the *TmDate* parameter, into a character string pointed to by the *String* parameter under the direction of the format string pointed to by the *Format* parameter. The actual values for the format specifiers are dependent on the current settings for the **LC_TIME** category. The **tm** structure values may be assigned by the user or generated by the **localtime** or **gmtime** subroutine. The resulting string is similar to the result of the **printf** *Format* parameter, and is placed in the memory location addressed by the *String* parameter. The maximum length of the string is determined by the *Length* parameter and terminates with a null character.

Many conversion specifications are the same as those used by the **date** command. The interpretation of some conversion specifications is dependent on the current locale of the process.

The *Format* parameter is a character string containing two types of objects: plain characters that are simply placed in the output string, and conversion specifications that convert information from the *TmDate* parameter into readable form in the output string. Each conversion specification is a sequence of this form: `% type`.

- A **%** (percent sign) introduces a conversion specification.
- The type of conversion is specified by one or two conversion characters. The characters and their meanings are:

Item	Description
%a	Represents the locale's abbreviated weekday name (for example, Sun) defined by the **abday** statement in the **LC_TIME** category.
%A	Represents the locale's full weekday name (for example, Sunday) defined by the **day** statement in the **LC_TIME** category.
%b	Represents the locale's abbreviated month name (for example, Jan) defined by the **abmon** statement in the **LC_TIME** category.
%B	Represents the locale's full month name (for example, January) defined by the **mon** statement in the **LC_TIME** category.
%c	Represents the locale's date and time format defined by the **d_t_fmt** statement in the **LC_TIME** category.

Item	Description
%C	Represents the century number (the year divided by 100 and truncated to an integer) as a decimal number (00 through 99 calculated from the standard term *tm_year* that is defined in the **tm** structure of the `time.h` library) If a minimum field width is not specified, the number of characters that are placed into the array pointed by the **string** parameter is the number of digits in the year divided by 100 or two, whichever is greater. If a minimum field width is specified, the number of characters that are placed into the array pointed by the **string** parameter is the number of digits in the year divided by 100 or the minimum field width, whichever is greater.
%d	Represents the day of the month as a decimal number (01 to 31).
%D	Represents the date in **%m/%d/%y** format (for example, `01/31/91`).
%e	Represents the day of the month as a decimal number (01 to 31). The **%e** field descriptor uses a two-digit field. If the day of the month is not a two-digit number, the leading digit is filled with a space character.
%E	Represents the locale's combined alternate era year and name, respectively, in **%o %N** format.
%F	Represents the date in **%+4Y-%m-%d** format if no flag and no minimum field width are specified (calculated from the standard terms *tm_year*, *tm_mon*, and *tm_mday* that are defined in the **tm** structure of the `time.h` library). If a minimum field width of less than or equal to 6 bytes is specified, the output string of the year is same as the output string by the Y specifier with any specified flag and a minimum field width of 6 bytes. If the specified field width is less than 6 bytes, the field width is considered as 6 bytes by default. If the minimum field width is specified as 10 bytes, and the year is four digits long, the output string that is produced matches the ISO 8601:2000 standard subclause 4.1.2.2 complete representation, extended format date representation of a specific day (YYYY-MM-DD). For example, 2021-05-20. If a + flag and a minimum field width of greater than 6 bytes is specified, and 7 bytes are sufficient to hold the digits of the year (not including any sign character), the output string matches the ISO 8601:2000 standard subclause 4.1.2.4 complete representation, extended format date representation of a specific day (±YYYYY-MM-DD). For example, +002021-05-20.
%G	Represents the week-based year as a decimal number, for example, 1977 (calculated from the standard terms *tm_year*, *tm_wday*, *tm_yday* that are defined in the **tm** structure of the `time.h` library). If a minimum field width is specified, the number of characters that are placed into the array pointed by the **string** parameter is the number of digits and leading sign characters (if any) in the year, or the minimum field width, whichever is greater.
%g	Represents the last two digit of ISO 8601 week-based year as a decimal number (0 to 99). It's like %G, but without century. (Calculated from tm_year, tm_yday, and tm_wday.)
%h	Represents the locale's abbreviated month name (for example, `Jan`) defined by the **abmon** statement in the **LC_TIME** category. This field descriptor is a synonym for the **%b** field descriptor.
%H	Represents the 24-hour-clock hour as a decimal number (00 to 23).
%I	Represents the 12-hour-clock hour as a decimal number (01 to 12).
%j	Represents the day of the year as a decimal number (001 to 366).
%k	Represents the 24-hour-clock hour clock as a right-justified space-filled number (0 to 23).
%m	Represents the month of the year as a decimal number (01 to 12).

Item	Description
%M	Represents the minutes of the hour as a decimal number (00 to 59).
%n	Specifies a new-line character.
%N	Represents the locale's alternate era name.
%o	Represents the alternate era year.
%p	Represents the locale's a.m. or p.m. string defined by the **am_pm** statement in the **LC_TIME** category.
%r	Represents 12-hour clock time with a.m./p.m. notation as defined by the **t_fmt_ampm** statement. The usual format is **%I:%M:%S %p**.
%R	Represents 24-hour clock time in **%H:%M** format.
%s	Represents the number of seconds since January 1, 1970, Coordinated Universal Time (CUT).
%S	Represents the seconds of the minute as a decimal number (00 to 59).
%t	Specifies a tab character.
%T	Represents 24-hour-clock time in the format **%H:%M:%S** (for example, 16:55:15).
%u	Represents the weekday as a decimal number (1 to 7). Monday or its equivalent is considered the first day of the week for calculating the value of this field descriptor.
%U	Represents the week of the year as a decimal number (00 to 53). Sunday, or its equivalent as defined by the **day** statement in the **LC_TIME** category, is considered the first day of the week for calculating the value of this field descriptor.
%V	Represents the week number of the ISO 8601 week-based year (with Monday as the first day of the week) as a decimal number (01 to 53). If the week containing January 1 has four or more days in the new year, then it is considered week 1; otherwise, it is considered week 52 (or 53 if the previous year was a leap year) of the previous year, and the next week is week 1 of the new year.
%w	Represents the day of the week as a decimal number (0 to 6). Sunday, or its equivalent as defined by the **day** statement, is considered as 0 for calculating the value of this field descriptor.
%W	Represents the week of the year as a decimal number (00 to 53). Monday, or its equivalent as defined by the **day** statement, is considered the first day of the week for calculating the value of this field descriptor.
%x	Represents the locale's date format as defined by the **d_fmt** statement.
%X	Represents the locale's time format as defined by the **t_fmt** statement.
%y	Represents the year of the century. **Note:** When the environment variable **XPG_TIME_FMT=ON**, **%y** is the year within the century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999, inclusive); values in the range 00-68 refer to 2000 to 2068, inclusive.
%Y	Represents the year as a decimal number, for example, 1989 (calculated from the standard term *tm_year*that is defined in the **tm** structure of the `time.h` library). If a minimum field width is specified, the number of characters that are placed into the array pointed by the **string** parameter is the number of digits and leading sign characters (if any) in the year, or the minimum field width, whichever is greater.

Item	Description
%z	Represents the offset from Coordinated Universal Time (UTC) in the ISO 8601 format **–0430** means 4 hours 30 minutes behind UTC, west of Greenwich, or by no characters if you cannot determine the time zone [tm_isdst]. **Note:** You must set the value of the XPG_SUS_ENV=ON environment variable to use the **%z** option else it falls back to the **%Z** option.
%Z	Represents the time-zone name if one can be determined (for example, EST). No characters are displayed if a time zone cannot be determined.
%%	Specifies a % (percent sign).

Some conversion specifiers can be modified by the **E** or **O** modifier characters to indicate that an alternative format or specification should be used. If the alternative format or specification does not exist for the current locale, the behavior will be the same as with the unmodified conversion specification. The following modified conversion specifiers are supported:

Item	Description
%Ec	Represents the locale's alternative appropriate date and time as defined by the **era_d_t_fmt** statement.
%EC	Represents the name of the base year (or other time period) in the locale's alternative form as defined by the **era** statement under the **era_name** category of the current era.
%Ex	Represents the locale's alternative date as defined by the **era_d_fmt** statement.
%EX	Represents the locale's alternative time as defined by the **era_t_fmt** statement.
%Ey	Represents the offset from the **%EC** modified conversion specifier (year only) in the locale's alternative form.
%EY	Represents the full alternative-year form.
%Od	Represents the day of the month, using the locale's alternative numeric symbols, filled as needed with leading 0's if an alternative symbol for 0 exists. If an alternative symbol for 0 does not exist, the **%Od** modified conversion specifier uses leading space characters.
%Oe	Represents the day of the month, using the locale's alternative numeric symbols, filled as needed with leading 0's if an alternative symbol for 0 exists. If an alternative symbol for 0 does not exist, the **%Oe** modified conversion specifier uses leading space characters.
%OH	Represents the hour in 24-hour clock time, using the locale's alternative numeric symbols.
%OI	Represents the hour in 12-hour clock time, using the locale's alternative numeric symbols.
%Om	Represents the month, using the locale's alternative numeric symbols.
%OM	Represents the minutes, using the locale's alternative numeric symbols.
%OS	Represents the seconds, using the locale's alternative numeric symbols.
%Ou	Represents the weekday as a number using the locale's alternative numeric symbols.
%OU	Represents the week number of the year, using the locale's alternative numeric symbols. Sunday is considered the first day of the week. Use the rules corresponding to the **%U** conversion specifier.
%OV	Represents the week number of the year (Monday as the first day of the week, rules corresponding to %V) using the locale's alternative numeric symbols.
%Ow	Represents the number of the weekday (with Sunday equal to 0), using the locale's alternative numeric symbols.

Item	Description
%OW	Represents the week number of the year using the locale's alternative numeric symbols. Monday is considered the first day of the week. Use the rules corresponding to the **%W** conversion specifier.
%Oy	Represents the year (offset from %C) using the locale's alternative numeric symbols.

The **strftime_l()** subroutine is similar to the **strftime()** subroutine, except the locale information that is specified in the locale variable. If the locale variable in the **strftime_l()** subroutine is set as a special locale object, LC_GLOBAL_LOCALE, or is not a valid locale object handle, the **strftime_l()** subroutine might result unexpected results.

Parameters

Item	Description
String	Points to the string to hold the formatted time.
Length	Specifies the maximum length of the string pointed to by the *String* parameter.
Format	Points to the format character string.
TmDate	Points to the time structure that is to be converted.
Locale	Points to the locale object that contains the locale information.

Return Values

If the total number of resulting bytes, including the terminating null byte, is not more than the *Length* value, the **strftime** subroutine returns the number of bytes placed into the array pointed to by the *String* parameter, not including the terminating null byte. Otherwise, a value of 0 is returned and the contents of the array are indeterminate.

strlen, , strnlen, strchr, strrchr, strpbrk, strspn, strcspn, strstr, strtok, or strsep Subroutine

Purpose

Determines the size, location, and existence of strings in memory.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>

size_t strlen (String)
const char *String;

size_t strnlen (String, maxlen)
const char *String;
size_t maxlen;

char *strchr (String, Character)
const char *String;
int Character;

char *strrchr (String, Character)
const char *String;
int Character;
```

```
char *strpbrk (String1, String2)
const char *String1, String2;

size_t strspn (String1, String2)
const char *String1, * String2;

size_t strcspn (String1, String2)
const char *String1, *String2;

char *strstr (String1, String2)
const char *String1, *String2;

char *strtok (String1, String2)
char *String1;
const char *String2;

char *strsep (String1, String2)
char **String1;
const char *String2;

char *index (String, Character)
const char *String;
int Character;

char *rindex (String, Character)
const char *String;
int Character;
```

Description

Attention: Do not use the **strtok** subroutine in a multithreaded environment. Use the **strtok_r** subroutine instead.

The **strlen**, **strnlen**, **strchr**, **strrchr**, **strpbrk**, **strspn**, **strcspn**, **strstr**, and **strtok** subroutines determine such values as size, location, and the existence of strings in memory.

The *String1*, *String2*, and *String* parameters point to strings. A string is an array of characters terminated by a null character.

The **strlen** subroutine returns the number of bytes in the string pointed to by the *String* parameter, not including the terminating null bytes.

The **strnlen** function returns an integer containing the smaller of either the length of the string pointed to by *String*, or *maxlen*, not including the terminating null bytes.

The **strchr** subroutine returns a pointer to the first occurrence of the character specified by the *Character* (converted to an unsigned character) parameter in the string pointed to by the *String* parameter. A null pointer is returned if the character does not occur in the string. The null byte that terminates a string is considered to be part of the string.

The **strrchr** subroutine returns a pointer to the last occurrence of the character specified by the *Character* (converted to a character) parameter in the string pointed to by the *String* parameter. A null pointer is returned if the character does not occur in the string. The null byte that terminates a string is considered to be part of the string.

The **strpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the *String1* parameter of any bytes from the string pointed to by the *String2* parameter. A null pointer is returned if no bytes match.

The **strspn** subroutine returns the length of the initial segment of the string pointed to by the *String1* parameter, which consists entirely of bytes from the string pointed to by the *String2* parameter.

The **strcspn** subroutine returns the length of the initial segment of the string pointed to by the *String1* parameter, which consists entirely of bytes *not* from the string pointed to by the *String2* parameter.

The **strstr** subroutine finds the first occurrence in the string pointed to by the *String1* parameter of the sequence of bytes specified by the string pointed to by the String2 parameter (excluding the terminating null character). It returns a pointer to the string found in the *String1* parameter, or a null pointer if the string was not found. If the *String2* parameter points to a string of 0 length, the **strstr** subroutine returns the value of the *String1* parameter.

The **strtok** subroutine breaks the string pointed to by the *String1* parameter into a sequence of tokens, each of which is delimited by a byte from the string pointed to by the *String2* parameter. The first call in the sequence takes the *String1* parameter as its first argument and is followed by calls that take a null pointer as their first argument. The separator string pointed to by the *String2* parameter may be different from call to call.

The first call in the sequence searches the *String1* parameter for the first byte that is not contained in the current separator string pointed to by the *String2* parameter. If no such byte is found, no tokens exist in the string pointed to by the *String1* parameter, and a null pointer is returned. If such a byte is found, it is the start of the first token.

The **strtok** subroutine then searches from the first token for a byte that is contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by the *String1* parameter, and subsequent searches for a token return a null pointer. If such a byte is found, the **strtok** subroutine overwrites it with a null byte, which terminates the current token. The **strtok** subroutine saves a pointer to the following byte, from which the next search for a token will start. The subroutine returns a pointer to the first byte of the token.

Each subsequent call with a null pointer as the value of the first argument starts searching from the saved pointer, using it as the first token. Otherwise, the subroutine's behavior does not change.

The **strsep** subroutine returns the next token from the string *String1* which is delimited by *String2*. The token is terminated with a `\0` character and *String1* is updated to point past the token. The **strsep** subroutine returns a pointer to the token, or NULL if *String2* is not found in *String1*.

The **index**, **rindex** and **strsep** subroutines are included for compatibility with BSD and are not part of the ANSI C Library. The **index** subroutine is implemented as a call to the **strchr** subroutine. The **rindex** subroutine is implemented as a call to the **strrchr** subroutine.

Parameters

Item	Description
Character	Specifies a character for which to return a pointer.
String	Points to a string from which data is returned.
String1	Points to a string from which an operation returns results.
String2	Points to a string which contains source for an operation.

Error Codes

The **strlen**, **strnlen**, **strchr**, **strrchr**, **strpbrk**, **strspn**, **strcspn**, **strstr**, and **strtok** subroutines fail if the following occurs:

Item	Description
EFAULT	A string parameter is an invalid address.

strncollen Subroutine

Purpose

Returns the number of collation values for a given string.

Library

Standard C Library (**libc.a**)

Syntax

```
include <string.h>

int strncollen ( String,  Number)
const char *String;
const int Number;
```

Description

The **strncollen** subroutine returns the number of collation values for a given string pointed to by the *String* parameter. The count of collation values is terminated when either a null character is encountered or when the number of bytes indicated by the *Number* parameter have been examined.

The collation values are set by the **setlocale** subroutine for the **LC_COLLATE** category. For example, if the locale is set to Es_ES (Spanish spoken in Spain) for the **LC_COLLATE** category, where `ch' has one collation value, then **strncollen** ('abchd', 5) returns 4.

In German, the <Sharp-S> character has two collation values, so substituting the <Sharp-S> character for B in the following example, **strncollen** `('straBa', 6)` returns 7.

If a character has no collation value, its collation length is 0.

Parameters

Item	Description
Number	The number of bytes in a string to be examined.
String	Pointer to a string to be examined for collation value.

Return Values

Upon successful completion, the **strncollen** subroutine returns the collation value for a given string, pointed to by the *String* parameter.

strtod32, strtod64, or strtod128 Subroutine

Purpose

Converts a string to a decimal floating-point number.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

_Decimal32 strtod32 (nptr, endptr)
const char *nptr;
char **endptr;

_Decimal64 strtod64 (nptr, endptr)
const char *nptr;
char **endptr;

_Decimal128 strtod128 (nptr, endptr)
const char *nptr;
char **endptr;
```

Description

The **strtod32**, **strtod64**, and **strtod128** subroutines convert the initial portion of the string pointed to by the *nptr* parameter to **_Decimal32**, **_Decimal64**, and **_Decimal128** representation, respectively. First, these subroutines decompose the input string into three parts:

- An initial and possibly empty sequence of white-space characters (as specified by the **isspace** subroutine)
- A subject sequence that is interpreted as a floating-point constant or represents infinity or NaN
- A final string of one or more unrecognized characters, including the terminating null byte of the input string

Then, the **strtod32**, **strtod64**, and **strtod128** subroutines attempt to convert the subject sequence to a floating-point number and return the result.

The expected form of the subject sequence is an optional plus or minus sign and one of the following:

- A non-empty sequence of decimal digits that might contain a radix character and an exponent part
- INF, INFINITY, or any other string equivalent except for case
- NAN or NAN (*n-char-sequence* $_{opt}$), ignoring case in the NAN, where:

```
n-char-sequence:
              digit
              n-char-sequence digit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence contains no characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the radix character (whichever occurs first) are interpreted as a floating constant according to the rules of the C language, except that the sequence is not a hexadecimal floating number or the radix character is used in place of a period. If neither an exponent part nor a radix character appears in a decimal floating-point number, an exponent part of the appropriate type with a value of 0 is assumed to follow the last digit in the string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A character sequence INF or INFINITY is interpreted as infinity. A character sequence NAN or NAN (*n-char-sequence* $_{opt}$) is interpreted as a quiet NaN. The meaning of the *n*-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by the *endptr* parameter, provided that the *endptr* parameter is not a null pointer.

The radix character is defined in the locale of the program (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period.

In locales other than the C or POSIX locale, other implementation-defined subject sequences can be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed. The value of the *nptr* parameter is stored in the object pointed to by the *endptr* parameter , provided that the *endptr* parameter is not a null pointer.

The **strtod32**, **strtod64**, and **strtod128** subroutines do not change the setting of the **errno** global variable if successful.

The value of 0 is returned on error and it is also a valid return value on success. Therefore, an application checking for error situations must set the value of the **errno** global variable to 0, call the **strtod32**, **strtod64**, or **strto128** subroutine, and check the **errno** global variable.

Note: Starting with the IBM AIX 6 with Technology Level 7 and the IBM AIX 7 with Technology Level 1, the precision of the floating-point conversion routines, printf and scanf family of functions has been increased from 17 digits to 37 digits for double and long double values.

Parameters

Item	Description
nptr	Contains a pointer to the string to be converted to a decimal floating point value.
endpr	Contains a pointer to the position in the string specified by the *nptr* parameter where a character is found that is not a valid character for the conversion.

Return Values

Upon successful completion, the **strtod32**, **strtod64**, and **strtod128** subroutines return the converted value. If no conversion can be performed, the value of 0 is returned and the **errno** global variable might be set to EINVAL.

If the correct value is outside the range of representable values, **±HUGE_VAL_D32**, **±HUGE_VAL_D64**, or **±HUGE_VAL_D128** is returned (according to the return type and sign of the value), and the **errno** global variable is set to **ERANGE**.

If the correct value causes underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type is returned, and the **errno** global variable is set to **ERANGE**.

strtof, strtod, or strtold Subroutine

Purpose

Converts a string to a double-precision number.

Syntax

```
#include <stdlib.h>

float strtof (nptr, endptr)
const char *restrict nptr;
char **restrict endptr;

double strtod ( nptr,  endptr)
const char *nptr
char**endptr;

long double strtold (nptr, endptr)
const char *restrict nptr;
char **restrict endptr;
```

Description

The **strtof**, **strtod**, and **strtold** subroutines convert the initial portion of the string pointed to by *nptr* to **double**, **float**, and **long double** representation, respectively. First, they decompose the input string into three parts:

- An initial, possibly empty, sequence of white-space characters (as specified by isspace()).
- A subject sequence interpreted as a floating-point constant or representing infinity or NaN.
- A final string of one or more unrecognized characters, including the terminating null byte of the input string.

Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, and one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, and an optional exponent part

- A 0x or 0X, and a non-empty sequence of hexadecimal digits optionally containing a radix character, and an optional binary exponent part
- One of INF or INFINITY, ignoring case
- One of NAN or NAN(*n-char-sequence* $_{opt}$), ignoring case in the NAN part, where:

```
n-char-sequence:
         digit
         nondigit
        n-char-sequence digit
        n-char-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of characters starting with the first digit or the decimal-point character (whichever occurs first) are interpreted as a floating constant of the C language, except that the radix character is used in place of a period, and if neither an exponent part nor a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A character sequence INF or INFINITY shall be interpreted as an infinity, if representable in the return type, or else as if it were a floating constant that is too large for the range of the return type. A character sequence NAN or NAN(*n-char-sequence* $_{opt}$) is interpreted as a quiet NaN, if supported in the return type, or else as if it were a subject sequence part that does not have the expected form. The meaning of the *n*-char sequences is implementation-defined. A pointer to the final string is stored in the object pointed to by the *endptr* parameter, provided that the *endptr* parameter is not a null pointer.

If the subject sequence has the hexadecimal form, the value resulting from the conversion is correctly rounded.

The radix character is defined in the program's locale (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period.

In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion shall be performed; the value of **str** is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The **strtod** subroutine does not change the setting of the **errno** global variable if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set errno to 0, call the **strtof** or **strtold** subroutine, then check **errno**.

Note: Starting with the IBM AIX 6 with Technology Level 7 and the IBM AIX 7 with Technology Level 1, the precision of the floating-point conversion routines, printf and scanf family of functions has been increased from 17 digits to 37 digits for double and long double values.

Parameters

Item	Description
nptr	Specifies the string to be converted.
endptr	Points to the final string.

Return Values

Upon successful completion, the **strtof** and **strtold** subroutines return the converted value. If no conversion could be performed, 0 is returned, and the **errno** global variable may be set to EINVAL.

If the correct value is outside the range of representable values, **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the sign of the value), and **errno** is set to ERANGE.

If the correct value would cause an underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type is returned and the **errno** global variable is set to ERANGE.

Error Codes

Note: Because a value of 0 can indicate either an error or a valid result, an application that checks for errors with the **strtod**, **strtof**, and **strtold** subroutines should set the **errno** global variable equal to 0 prior to the subroutine call. The application can check the **errno** global variable after the subroutine call.

If the string pointed to by *NumberPointer* is empty or begins with an unrecognized character, a value of 0 is returned for the **strtod**, **strtof**, and **strtold** subroutines.

If the conversion cannot be performed, a value of 0 is returned, and the **errno** global variable is set to indicate the error.

If the conversion causes an overflow (that is, the value is outside the range of representable values), **+/- HUGE_VAL** is returned with the sign indicating the direction of the overflow, and the **errno** global variable is set to **ERANGE**.

If the conversion would cause an underflow, a properly signed value of 0 is returned and the **errno** global variable is set to **ERANGE**.

For the **strtod**, **strtof**, and **strtold** subroutines, if the value of the *EndPointer* parameter is not (**char****) NULL, a pointer to the character that stopped the subroutine is stored in **EndPointer*. If a floating-point value cannot be formed, **EndPointer* is set to *NumberPointer.*

The **strtof** subroutine has only one rounding error. (If the **strtod** subroutine is used to create a double-precision floating-point number and then that double-precision number is converted to a floating-point number, two rounding errors could occur.)

strtoimax or strtoumax Subroutine

The **strtoimax** and **strtoumax** subroutines return the converted value, if any.

If no conversion could be performed, zero is returned.

If the correct value is outside the range of representable values, {**INTMAX_MAX**}, {**INTMAX_MIN**}, or {**UINTMAX_MAX**} is returned (according to the return type and sign of the value, if any), and the **errno** global variable is set to ERANGE.

Purpose

Converts string to integer type.

Syntax

```
#include <inttypes.h>

intmax_t strtoimax (nptr, endptr, base)
const char *restrict nptr;
char **restrict endptr;
int base;

uintmax_t strtoumax (nptr, endptr, base)
const char *restrict nptr;
char **restrict endptr;
int base;
```

Description

The **strtoimax** and **strtoumax** subroutines are equivalent to the **strtol**, **strtoll**, **strtoul**, and **strtoull** subroutines, except that the initial portion of the string shall be converted to **intmax_t** and **uintmax_t** representation, respectively.

Parameters

Item	Description
nptr	Points to the string to be converted.
endptr	Points to the object where the final string is stored.
base	Determines the value of the integer represented in some radix.

Return Values

strtok_r Subroutine

Purpose

Breaks a string into a sequence of tokens.

Libraries

Thread-Safe C Library (**libc_r.a**)

Syntax

```
#include<string.h>
char *strtok_r (String, Separators, Pointer);
char *String;
const char *Separators;
char **Pointer;
```

Description

Note: The **strtok_r** subroutine is used in a multithreaded environment.

The **strtok_r** subroutine breaks the string pointed to by the *String* parameter into a sequence of tokens, each of which is delimited by a byte from the string pointed to by the *Separators* parameter. The *Pointer* parameter holds the information necessary for the **strtok_r** subroutine to perform scanning on the *String* parameter. In the first call to the **strtok_r** subroutine, the value passed as the *Pointer* parameter is ignored.

The first call in the sequence searches the *String* parameter for the first byte that is not contained in the current separator string pointed to by the *Separators* parameter. If no such byte is found, no tokens exist in the *String* parameter, and a null pointer is returned. If such a byte is found, it is the start of the first token. The **strtok_r** subroutine also updates the *Pointer* parameter with the starting address of the token following the first occurrence of the *Separators* parameter.

In subsequent calls, a null pointer should be passed as the first parameter to the **strtok_r** subroutine instead of the *String* parameter. Each subsequent call with a null pointer as the value of the first argument starts searching from the *Pointer* parameter, using it as the first token. Otherwise, the subroutine's behavior does not change. The **strtok_r** subroutine would return successive tokens until no tokens remain. The *Separators* parameter may be different from one call to another.

Parameters

Item	Description
String	Points to a string from which an operation returns results.
Separators	Points to a string which contains source for an operation.
Pointer	Points to a user provided pointer.

Error Codes

The **strtok_r** subroutine fails if the following occurs:

Item	Description
EFAULT	A *String* parameter is an invalid address.

strtol, strtoul, strtoll, strtoull, or atoi Subroutine

Purpose

Converts a string to a signed or unsigned long integer or long long integer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
long strtol ( String,  EndPointer,  Base)
const char *String;
char **EndPointer;
int Base;
```

```
unsigned long strtoul (String, EndPointer, Base)
const char *String;
char **EndPointer;
int Base;
```

```
long long int strtoll (String, EndPointer, Base)
char *String, **EndPointer;
int Base;
```

```
unsigned long long int strtoull (String, EndPointer, Base)
char *String, **EndPointer;
int Base;
```

```
int atoi (String)
const char *String;
```

Description

The **strtol** subroutine returns a long integer whose value is represented by the character string to which the *String* parameter points. The **strtol** subroutine scans the string up to the first character that is

inconsistent with the *Base* parameter. Leading white-space characters are ignored, and an optional sign may precede the digits.

The **strtoul** subroutine provides the same functions but returns an unsigned long integer.

The **strtoll** and **strtoull** subroutines provide the same functions but return long long and unsigned long long integers, respectively.

The **atoi** subroutine is equivalent to the **strtol** subroutine where the value of the *EndPointer* parameter is a null pointer and the *Base* parameter is a value of 10.

If the value of the *EndPointer* parameter is not null**,** then a pointer to the character that ended the scan is stored in *EndPointer*. If an integer cannot be formed, the value of the *EndPointer* parameter is set to that of the *String* parameter.

If the *Base* parameter is a value between 2 and 36, the subject sequence's expected form is a sequence of letters and digits representing an integer whose radix is specified by the *Base* parameter. This sequence is optionally preceded by a + (positive) or - (negative) sign. Letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the *Base* parameter are permitted. If the *Base* parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the + (positive) or - (negative) sign if present.

If the value of the *Base* parameter is 0, the string determines the base. Thus, after an optional leading sign, a leading 0 indicates octal conversion, and a leading 0x or 0X indicates hexadecimal conversion. The default is to use decimal conversion.

Parameters

Item	Description
String	Points to the character string to be converted.
EndPointer	Points to a character string that contains the first character not converted.
Base	Specifies the base to use for the conversion.

Return Values

Upon successful completion, the **strtol**, **strtoul**, **strtoll**, and **strtoull** subroutines return the converted value. If no conversion could be performed, 0 is returned, and the **errno** global variable is set to indicate the error. If the correct value is outside the range of representable values, the **strtol** subroutine returns a value of **LONG_MAX** or **LONG_MIN** according to the sign of the value, while the **strtoul** subroutine returns a value of **ULONG_MAX**. The **strtoll** subroutine returns a value of **LLONG_MAX** or **LLONG_MIN**, according to the sign of the value. The **strtoul** subroutine returns a value of **ULONG_MAX**, and the **strtoull** subroutine returns a value of **ULLONG_MAX**.

Error Codes

The **strtol** and **strtoul** subroutines return the following error codes:

Item	Description
ERANGE	The correct value of the converted number causes underflow or overflow.
EINVAL	The value of the *Base* parameter is not valid.

strptime Subroutine

Purpose

Converts a character string to a time value.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>
```

```
char *strptime ( Buf,  Format,  Tm)
const char *Buf, *Format;
struct tm *Tm;
```

Description

The **strptime** subroutine converts the characters in the *Buf* parameter to time values that are stored in the *Tm* structure by using the format specified by the *Format* parameter.

The *Format* parameter can contain of zero or more specifiers. Each specifier consists of one of the following elements:

- One or more white-space characters as specified by the **isspace** subroutine.
- An ordinary character that is neither a percent sign (%) character nor a white-space character.
- A conversion specification can be introduced by a percent sign (%). The following items can be included in a sequence after the conversion specification:
  - An optional flag, the zero character (0) or the plus sign (+) is ignored.
  - An optional width of the field descriptor. If a width of the field descriptor is specified, it is read as a string of decimal digits that determines the maximum number of bytes that are converted rather than the number of bytes that are specified by the conversion specifiers.

Parameters

Item	Description
Buf	Contains the character string to be converted by the **strptime** subroutine.
Format	Contains format specifiers for the **strptime** subroutine. The *Format* parameter contains 0 or more specifiers. Each specifier is composed of one of the following elements: • One or more white-space characters • An ordinary character (neither % [percent sign] nor a white-space character) • A format specifier

Note: If more than one format specifier is present, they must be separated by white space or a non-percent/non-alphanumeric character. If the seperator between format specifiers is other than white space, the *Buf* string should hold the same seperator at the corresponding locations.

The **LC_TIME** category defines the locale values for the format specifiers. The following format specifiers are supported:

Item	Description
%a	Represents the weekday name, either abbreviated as specified by the **abday** statement or full as specified by the **day** statement.
%A	Represents the weekday name, either abbreviated as specified by the **abday** statement or full as specified by the **day** statement.

Item	Description
%b	Represents the month name, either abbreviated as specified by the **abmon** statement or full as specified by the **month** statement.
%B	Represents the month name, either abbreviated as specified by the **abmon** statement or full as specified by the **month** statement.
%c	Represents the date and time format defined by the **d_t_fmt** statement in the **LC_TIME** category.
%C	Represents the first two-digits of the year. Leading zeros are permitted but it is not required. Also, a leading plus sign (+) or minus sign (-) is permitted before any leading zeros but it is not required.
%d	Represents the day of the month as a decimal number (01 to 31).
%D	Represents the date in **%m/%d/%y** format (for example, 01/31/91).
%e	Represents the day of the month as a decimal number (01 to 31).
%E	Represents the combined alternate era year and name, respectively, in **%o %N** format.
%h	Represents the month name, either abbreviated as specified by the **abmon** statement or full as specified by the **month** statement.
%H	Represents the 24-hour-clock hour as a decimal number (00 to 23).
%I	Represents the 12-hour-clock hour as a decimal number (01 to 12).
%j	Represents the day of the year as a decimal number (001 to 366).
%m	Represents the month of the year as a decimal number (01 to 12).
%M	Represents the minutes of the hour as a decimal number (00 to 59).
%n	Represents any white space.
%N	Represents the alternate era name.
%o	Represents the alternate era year.
%p	Represents the a.m. or p.m. string defined by the **am_pm** statement in the **LC_TIME** category.
%r	Represents 12-hour-clock time with a.m./p.m. notation as defined by the **t_fmt_ampm** statement, usually in the format **%I:%M:%S %p**.
%S	Represents the seconds of the minute as a decimal number (00 to 61). The decimal number range of 00 to 61 provides for leap seconds.
%t	Represents any white space.
%T	Represents 24-hour-clock time in the format **%H:%M:%S** (for example, 16 : 55 : 15).
%U	Represents the week of the year as a decimal number (00 to 53). Sunday, or its equivalent as defined by the **day** statement, is considered the first day of the week for calculating the value of this field descriptor.
%w	Represents the day of the week as a decimal number (0 to 6). Sunday, or its equivalent as defined by the **day** statement in the **LC_TIME** category, is considered to be 0 for calculating the value of this field descriptor.
%W	Represents the week of the year as a decimal number (00 to 53). Monday, or its equivalent as defined by the **day** statement in the **LC_TIME** category, is considered the first day of the week for calculating the value of this field descriptor.
%x	Represents the date format defined by the **d_fmt** statement in the **LC_TIME** category.
%X	Represents the time format defined by the **t_fmt** statement in the **LC_TIME** category.

Item	Description
%y	Represents the last two-digits of the year. When the *Format* parameter contains neither a **%C** conversion specifier nor a **%Y** conversion specifier, the value in the range 69 - 99 refers to a year in the range 1969 - 1999, and the value in the range 00 - 68 refers to a year in the range 2000 - 2068. Leading zeros are permitted but it is not required. Also, a leading plus sign (+) or minus sign (-) is permitted before any leading zeros but it is not required.
%Y	Represents the four-digits of the year. Leading zeros are permitted but it is not required. Also, a leading plus sign (+) or minus sign (-) is permitted before any leading zeros but it is not required.
%Z	Represents the time-zone name, if one can be determined (for example, EST). No characters are displayed if a time zone cannot be determined.
%%	Specifies a % (percent sign) character.

Some format specifiers can be modified by the **E** and **O** modifier characters to indicate an alternative format or specification. If the alternative format or specification does not exist in the current locale, the behavior will be as if the unmodified format specifier were used. The following modified format specifiers are supported:

Item	Description
%Ec	Represents the locale's alternative appropriate date and time as defined by the **era_d_t_fmt** statement.
%EC	Represents the base year (or other time period) in the locale's alternative form as defined by the **era** statement under the **era_name** category of the current era.
%Ex	Represents the alternative date as defined by the **era_d_fmt** statement.
%EX	Represents the locale's alternative time as defined by the **era_t_fmt** statement.
%Ey	Represents the offset from the **%EC** format specifier (year only) in the locale's alternative form.
%EY	Represents the full alternative-year format.
%Od	Represents the month using the locale's alternative numeric symbols. Leading 0's are permitted but not required.
%Oe	Represents the month using the locale's alternative numeric symbols. Leading 0's are permitted but not required.
%OH	Represents the hour in 24-hour-clock time using the locale's alternative numeric symbols.
%OI	Represents the hour in 12-hour-clock time using the locale's alternative numeric symbols.
%Om	Represents the month using the locale's alternative numeric symbols.
%OM	Represents the minutes using the locale's alternative numeric symbols.
%OS	Represents the seconds using the locale's alternative numeric symbols.
%OU	Represents the week number of the year using the locale's alternative numeric symbols. Sunday is considered the first day of the week. Use the rules corresponding to the **%U** format specifier.
%Ow	Represents the day of the week using the locale's alternative numeric symbols. Sunday is considered the first day of the week.
%OW	Represents the week number of the year using the locale's alternative numeric symbols. Monday is considered the first day of the week. Use the rules corresponding to the **%W** format specifier.
%Oy	Represents the year (offset from %C) using the locale's alternative numeric symbols.

A format specification consisting of white-space characters is performed by reading input until the first nonwhite-space character (which is not read) or up to no more characters can be read.

A format specification consisting of an ordinary character is performed by reading the next character from the *Buf* parameter. If this character differs from the character comprising the directive, the directive fails and the differing character and any characters following it remain unread. Case is ignored when matching *Buf* items, such as month or weekday names.

A series of directives composed of **%n** format specifiers, **%t** format specifiers, white-space characters, or any combination of the three items is processed by reading up to the first character that is not white space (which remains unread), or until no more characters can be read.

Item	Description
Tm	Specifies the structure to contain the output of the **strptime** subroutine. If a conversion fails, the contents of the *Tm* structure are undefined.

Return Values

If successful, the **strptime** subroutine returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

stty or gtty Subroutine

Purpose

Sets or gets terminal state.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sgtty.h>
```

```
stty ( FileDescriptor,  Buffer)
int FileDescriptor;
struct sgttyb *Buffer;
```

```
gtty (FileDescriptor, Buffer)
int FileDescriptor;
struct sgttyb *Buffer;
```

Description

These subroutines have been made obsolete by the **ioctl** subroutine.

The **stty** subroutine sets the state of the terminal associated with the *FileDescriptor* parameter. The **gtty** subroutine retrieves the state of the terminal associated with *FileDescriptor*. To set the state of a terminal, the calling process must have write permission.

Use of the **stty** subroutine is equivalent to the **ioctl (***FileDescriptor***,** TIOSETP**,** *Buffer***)** subroutine, while use of the **gtty** subroutine is equivalent to the **ioctl (***FileDescriptor***,** TIOGETP**,** *Buffer***)** subroutine.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.
Buffer	Specifies the buffer.

Return Values

If the **stty** or **gtty** subroutine is successful, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

subpad Subroutine

Purpose

Creates a subwindow within a pad.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

WINDOW *subpad(Orig,  NLines, NCols,  Begin_Y, Begin_X)
WINDOW * Orig;
int  NCols, NLines, Begin_Y,  Begin_X;
```

Description

The **subpad** subroutine creates and returns a pointer to a subpad. A subpad is a window within a pad. You specify the size of the subpad by supplying a starting coordinate and the number of rows and columns within the subpad. Unlike the **subwin** subroutine, the starting coordinates are relative to the pad and not the terminal's display.

Changes to the subpad affect the character image of the parent pad, as well. If you change a subpad, use the **touchwin** or **touchline** subroutine on the parent pad before refreshing the parent pad. Use the **prefresh** subroutine to refresh a pad.

Parameters

Item	Description
Orig	Points to the parent pad.
NLines	Specifies the number of lines (rows) in the subpad.
NCols	Specifies the number of columns in the subpad.
Begin_Y	Identifies the upper left-hand row coordinate of the subpad relative to the parent pad.
Begin_X	Identifies the upper left-hand column coordinate of the subpad relative to the parent pad.

Examples

To create a subpad, use:

```
WINDOW *orig, *mypad;
```

```
orig = newpad(100, 200);
```

```
mypad = subpad(orig, 30, 5, 25, 180);
```

The parent pad is 100 lines by 200 columns. The subpad is 30 lines by 5 columns and starts in line 25, column 180 of the parent pad.

subwin Subroutine

Purpose

Creates a subwindow within an existing window.

Library

Curses Library (**libcurses.a**)

Syntax

#include <curses.h> WINDOW *subwin (*ParentWindow***,** *NumLines***,** *NumCols***,***Line***,***Column***) WINDOW *** *ParentWindow* **; int** *NumLines***,** *NumCols***,** *Line***,** *Column***;**

Description

The **subwin** subroutine creates a subwindow within an existing window. You must supply coordinates for the subwindow relative to the terminal's display. Recall that the subwindow shares its parent's window buffer. Changes made to the shared window buffer in the area covered by a subwindow, through either the parent window or any of its subwindows, affects all windows sharing the window buffer.

When changing the image of a subwindow, it is necessary to call the **touchwin** ("touchwin Subroutine" on page 2180) or **touchline** subroutine on the parent window before calling the **wrefresh** ("refresh or wrefresh Subroutine" on page 1728) subroutine on the parent window.

Changes to one window will affect the character image of both windows.

Parameters

Item	Description
NumCols	Indicates the number of vertical columns in the subwindow's width. If 0 is passed as the *NumCols* value, the subwindow runs from the Column to the right edge of its parent window.
NumLines	Indicates the number of horizontal lines in the subwindow's height. If 0 is passed as the *NumLines* parameter, then the subwindow runs from the Line to the bottom of its parent window.
ParentWindow	Specifies the subwindow's parent.
Column	Specifies the horizontal coordinate for the upper-left corner of the subwindow. This coordinate is relative to the (0, 0) coordinates of the terminal, not the (0, 0) coordinates of the parent window. **Note:** The upper-left corner of the terminal is referenced by the coordinates (0, 0).
Line	Specifies the vertical coordinate for the upper-left corner of the subwindow. This coordinate is relative to the (0, 0) coordinates of the terminal, not the (0, 0) coordinates of the parent window. **Note:** The upper-left corner of the terminal is referenced by the coordinates (0, 0).

Return Values

When the **subwin** subroutine is successful, it returns a pointer to the subwindow structure. Otherwise, it returns the following:

Item	Description
ERR	Indicates one or more of the parameters is invalid or there is insufficient storage available for the new structure.

Examples

1. To create a subwindow, use:

```
WINDOW *my_window, *my_sub_window;

my_window = newwin (“derwin, newwin, or subwin Subroutine” on page 246)
                (5, 10, 20, 30);

my_sub_window = subwin(my_window, 2, 5, 20, 30);
```

   my_sub_window is now a subwindow 2 lines deep, 5 columns wide, starting at the same coordinates of its parent window my_window. That is, the subwindow's upper-left corner is at coordinates y = 20, x = 30 and lower-right corner is at coordinates y = 21, x = 34.

2. To create a subwindow that is flush with the right side of its parent, use:

```
WINDOW *my_window, *my_sub_window;

my_window = newwin (“derwin, newwin, or subwin Subroutine” on page 246)
                (5, 10, 20, 30);

my_sub_window = subwin(my_window, 2, 0, 20, 30);
```

   my_sub_window is now a subwindow 2 lines deep, extending all the way to the right side of its parent window my_window, and starting at the same coordinates. That is, the subwindow's upper-left corner is at coordinates y = 20, x = 30 and lower-right corner is at coordinates y = 21, x = 39.

3. To create a subwindow in the lower-right corner of its parent, use:

```
WINDOW *my_window, *my_sub_window

my_window = newwwin (“derwin, newwin, or subwin Subroutine” on page 246)
             (5, 10, 20, 30);

my_sub_window = subwin(my_window, 0, 0, 22, 35);
```

   my_sub_window is now a subwindow that fills the bottom right corner of its parent window, my_window, starting at the coordinates y = 22, x = 35. That is, the subwindow's upper-left corner is at coordinates y = 22, x = 35 and lower-right corner is at coordinates y = 24, x = 39.

swab Subroutine

Purpose

Copies bytes.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
void swab ( From,  To,  NumberOfBytes)
const void *From;
void *To;
ssize_t NumberOfBytes;
```

Description

The **swab** subroutine copies the number of bytes pointed to by the *NumberOfBytes* parameter from the location pointed to by the *From* parameter to the array pointed to by the *To* parameter, exchanging adjacent even and odd bytes.

The *NumberOfBytes* parameter should be even and nonnegative. If the *NumberOfBytes* parameter is odd and positive, the **swab** subroutine uses *NumberOfBytes* -1 instead. If the *NumberOfBytes* parameter is negative, the **swab** subroutine does nothing.

Parameters

Item	Description
From	Points to the location of data to be copied.
To	Points to the array to which the data is to be copied.
NumberOfBytes	Specifies the number of even and nonnegative bytes to be copied.

swapoff Subroutine

Purpose

Deactivates paging or swapping to a designated block device.

Library

Standard C Library (**libc.a**)

Syntax

```
int swapoff (PathName)
char *PathName;
```

Description

The **swapoff** subroutine deactivates a block device or logical volume that is actively being used for paging and swapping. There must be sufficient space to satisfy the system's paging space requirements in the remaining devices after this device is deactivated or **swapoff** will fail. Sufficient space must accommodate the current system-wide paging space usage and the npswarn value. Refer to the swap command for information on current system-wide paging space usage. Refer to the npswarn tunable parameter of the vmo command, and Values for the npswarn and npskill paramaters for information on the npswarn value.

Parameters

Item	Description
PathName	Specifies the full path name of the block device or logical volume.

Error Codes

If an error occurs, the **errno** global variable is set to indicate the error:

Item	Description
EBUSY	The deactivation is already running.
EINTR	The signal was received during the processing of a request.
ENODEV	The *PathName* file does not exist.
ENOMEM	No memory is available.
ENOSPC	There is not enough space in other paging spaces to satisfy the system's requirements.
ENOTBLK	The device must be a block device or logical volume.
ENOTDIR	A component of the *PathName* prefix is not a directory.
EPERM	Caller does not have proper authority.

Other errors are from calls to the device driver's **open** subroutine or **ioctl** subroutine.

swapon Subroutine

Purpose

Activates paging or swapping to a designated block device.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/vminfo.h>

int swapon ( PathName)
char *PathName;
```

Description

The **swapon** subroutine makes the designated block device available to the system for allocation for paging and swapping.

The specified block device must be a logical volume on a disk device. The paging space size is determined from the current size of the logical volume.

Parameters

Item	Description
PathName	Specifies the full path name of the block device.

Error Codes

If an error occurs, the **errno** global variable is set to indicate the error:

Item	Description
EINTR	Signal was received during processing of a request.
EINVAL	Invalid argument (size of device is invalid).
ENOENT	The *PathName* file does not exist.

Item	Description
ENOMEM	The maximum number of paging space devices (16) are already defined, or no memory is available.
ENOTBLK	Block device required.
ENOTDIR	A component of the *PathName* prefix is not a directory.
ENXIO	No such device address.

Other errors are from calls to the device driver's **open** subroutine or **ioctl** subroutine.

swapqry Subroutine

Purpose

Returns paging device status.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/vminfo.h>

int swapqry (PathName,  Buffer)
char *PathName;
struct pginfo *Buffer;
```

Description

The **swapqry** subroutine returns information to a user-designated buffer about active paging and swap devices.

Parameters

Item	Description
PathName	Specifies the full path name of the block device.
Buffer	Points to the buffer into which the status is stored.

Return Values

The **swapqry** subroutine returns 0 if the *PathName* value is an active paging device. If the *Buffer* value is not null, it also returns status information.

Error Codes

If an error occurs, the subroutine returns -1 and the **errno** global variable is set to indicate the error, as follows:

Item	Description
EFAULT	Buffer pointer is invalid.
EINVAL	Invalid argument.
EINTR	Signal was received while processing request.

Item	Description
ENODEV	Device is not an active paging device.
ENOENT	The *PathName* file does not exist.
ENOTBLK	Block device required.
ENOTDIR	A component of the *PathName* prefix is not a directory.
ENXIO	No such device address.

symlink or symlinkat Subroutine

Purpose

Makes a symbolic link to a file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int symlink ( Path1,  Path2)
const char *Path1;
const  char *Path2;

int symlinkat ( Path1, DirFileDescriptor, Path2)
const char * Path1;
int DirFileDescriptor;
const  char * Path2;
```

Description

The **symlink** and **symlinkat** subroutines create a symbolic link with the file named by the *Path2* parameter, which refers to the file named by the *Path1* parameter.

As with a hard link (described in the **link** subroutine), a symbolic link allows a file to have multiple names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link provides no such assurance. In fact, the file named by the *Path1* parameter need not exist when the link is created. In addition, a symbolic link can cross file system boundaries.

When a component of a path name refers to a symbolic link rather than a directory, the path name contained in the symbolic link is resolved. If the path name in the symbolic link starts with a / (slash), it is resolved relative to the root directory of the process. If the path name in the symbolic link does not start with / (slash), it is resolved relative to the directory that contains the symbolic link.

If the symbolic link is not the last component of the original path name, remaining components of the original path name are resolved from the symbolic-link point.

If the last component of the path name supplied to a subroutine refers to a symbolic link, the symbolic link path name may or may not be traversed. Most subroutines always traverse the link; for example, the **chmod**, **chown**, **link**, and **open** subroutines. The **statx** subroutine takes an argument that determines whether the link is to be traversed.

The following subroutines refer only to the symbolic link itself, rather than to the object to which the link refers:

Item	Description
mkdir	Fails with the **EEXIST** error code if the target is a symbolic link.
mknod	Fails with the **EEXIST** error code if a symbolic link exists with the same name as the target file as specified by the *Path* parameter in the **mknod** and **mkfifo** subroutines.
open	Fails with **EEXIST** error code when the **O_CREAT** and **O_EXCL** flags are specified and a symbolic link exists for the path name specified.
readlink ("readlink or readlinkat Subroutine" on page 1721)	Applies only to symbolic links.
rename ("rename or renameat Subroutine" on page 1743)	Renames the symbolic link if the file to be renamed (the *FromPath* parameter for the **rename** subroutine) is a symbolic link. If the new name (the *ToPath* parameter for the **rename** subroutine) refers to an existing symbolic link, the symbolic link is destroyed.
rmdir ("rmdir Subroutine" on page 1752)	Fails with the **ENOTDIR** error code if the target is a symbolic link.
symlink	Running this subroutine causes an error if a symbolic link named by the *Path2* parameter already exists. A symbolic link can be created that refers to another symbolic link; that is, the *Path1* parameter can refer to a symbolic link.
unlink ("unlink or unlinkat Subroutine" on page 2264)	Removes the symbolic link.

Since the mode of a symbolic link cannot be changed, its mode is ignored during the lookup process. Any files and directories referenced by a symbolic link are checked for access normally.

The **symlinkat** subroutine is equivalent to the **symlink** subroutine if the *DirFileDescriptor* parameter is set to **AT_FDCWD** or if the *Path2* parameter is an absolute path name. If the *DirFileDescriptor* parameter is a valid file descriptor of an open directory and the *Path2* parameter is a relative path name, the *Path2* parameter is considered as the relative path to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory.

If the *DirFileDescriptor* parameter is opened without the **O_SEARCH** open flag, the subroutine checks whether directory searches are permitted for that directory using the current permissions of the directory. If the directory is opened with the **O_SEARCH** open flag, the subroutine does not perform the check for that directory.

Parameters

Item	Description
Path1	Specifies the contents of the *Path2* symbolic link. This value is a null-terminated string representing the object to which the symbolic link will point. *Path1* cannot be the null value and cannot be more than **PATH_MAX** characters long. **PATH_MAX** is defined in the **limits.h** file.
DirFileDescriptor	Specifies the file descriptor of an open directory.

Item	Description
Path2	Names the symbolic link to be created. If *DirFileDescriptor* is specified and *Path2* is a relative path name, then *Path2* is considered relative to the directory specified by *DirFileDescriptor*.

Return Values

Upon successful completion, the **symlink** and **symlinkat** subroutines return a value of 0. If the **symlink** or the **symlinkat** subroutine fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **symlink** and **symlinkat** subroutines fail if one or more of the following are true:

Item	Description
EEXIST	*Path2* already exists.
EACCES	The requested operation requires writing in a directory with a mode that denies write permission.
EROFS	The requested operation requires writing in a directory on a read-only file system.
ENOSPC	The directory in which the entry for the symbolic link is being placed cannot be extended because there is no space left on the file system containing the directory.
EDQUOT	The directory in which the entry for the new symbolic link is being placed cannot be extended or disk blocks could not be allocated for the symbolic link because the user's or group's quota of disk blocks on the file system containing the directory has been exhausted.

The **symlinkat** subroutine is unsuccessful if one or more of the following settings are true:

Item	Description
EBADF	The *Path2* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a valid file descriptor.
ENOTDIR	The *Path2* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.

The **symlink** and **symlinkat** subroutines can be unsuccessful for other reasons.

sync Subroutine

Purpose

Updates all file systems.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

void sync ( )
```

Description

The **sync** subroutine causes all information in memory that should be on disk to be written out. The writing, although scheduled, is not necessarily complete upon return from this subroutine. Types of information to be written include modified superblocks, i-nodes, data blocks, and indirect blocks.

The **sync** subroutine should be used by programs that examine a file system, such as the **df** and **fsck** commands.

If Network File System (NFS) is installed on your system, information in memory that relates to remote files is scheduled to be sent to the remote node.

syncvfs Subroutine

Purpose

Updates a filesystem.

Syntax

```
#include <fscntl.h>

int syncvfs (vfsName, command)
char *vfsName;
int command;
```

Description

The **syncvfs** subroutine behaves in 3 different manners depending on the granularity specified. In each case the **GFS_SYNCVFS** flag is checked and **VFS_SYNCVFS** or **VFS_SYNC** is called on the GFS and/or VFS specified. In each case the the *command* parameter is passed untouched. The cases are:

- If a NULL pointer is passed through the *vfsName* parameter, the **FS_SYNCVFS_ALL** level is assumed, and the call loops through each GFS in a similar manner to the sync call.
- If **FS_SYNCVFS_FSTYPE** is passed, the GFS is scanned and the names compared. The GFS with the correct name (if one exists) is called with its own GFS pointer and a null VFS pointer.
- If **FS_SYNCVFS_FS** is passed, the mount point is looked up and, if it exists, **VFS_SYNCVFS** is called with the GFS pointer and the VFS pointer of the filesystem found.

Parameters

Item	Description
vfsName	Depending on the value of the *command* parameter, this can either be NULL, the name of a filesystem type (for example, "jfs", "j2") or the name of a filesystem, specified by mount point (for example, "/testj2").

Item	Description
command	Command is the mask of two options, a level and a granularity. The granularity can be one of: **FS_SYNCVFS_ALL** sync every filesystem **FS_SYNCVFS_FSTYPE** sync every filesystem of VFS type corresponding to *vfsName* **FS_SYNCVFS_FS** sync specific filesystem at *vfsName* The level can be one of: **FS_SYNCVFS_TRY** daemon heurstics **FS_SYNCVFS_FORCE** user requested sync **FS_SYNCVFS_QUIESCE** full filesystem quiesce

Return Values

Upon successful completion, the **syncvfs** subroutine returns 0. If unsuccessful, -1 is returned and the **errno** global variable is set.

_sync_cache_range Subroutine

Purpose

Synchronizes the I cache with the D cache.

Library

Standard C Library (**libc.a**)

Syntax

```
void _sync_cache_range (eaddr, count)
caddr_t  eaddr;
uint  count;
```

Description

The **_sync_cache_range** subroutine synchronizes the I cache with the D cache, given an effective address and byte count. Programs performing instruction modification can call this routine to ensure that the most recent instructions are fetched for the address range.

Parameters

Item	Description
eaddr	Specifies the starting effective address of the address range.
count	Specifies the byte count of the address range.

sysconf Subroutine

Purpose

Determines the current value of a specified system limit or option.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

long int sysconf (*Name***)**
int *Name***;**

Description

The **sysconf** subroutine determines the current value of certain system parameters, the configurable system limits, or whether optional features are supported. The *Name* parameter represents the system variable to be queried.

Parameters

Item	Description
Name	Specifies which system variable setting should be returned. The valid values for the *Name* parameter are defined in the **limits.h**, **time.h**, and **unistd.h** files and are described below:

Item	Description
_SC_AIO_LISTIO_MAX	Maximum number of Input and Output operations that can be specified in a list Input and Output call.
_SC_AIO_MAX	Maximum number of outstanding asynchronous Input and Output operations.
_SC_AIX_ENHANCED_AFFINITY	Determines if the ENHANCED_AFFINITY services are enabled.
_SC_ASYNCHRONOUS_IO	Implementation supports the Asynchronous Input and Output option.
_SC_ARG_MAX	Specifies the maximum byte length of the arguments for one of the **exec** functions, including environment data.
_SC_BC_BASE_MAX	Specifies the maximum number **ibase** and **obase** variables allowed by the ../b_commands/bc.html
_SC_BC_DIM_MAX	Specifies the maximum number of elements permitted in an array by the **bc** command.
_SC_BC_SCALE_MAX	Specifies the maximum **scale** variable allowed by the **bc** command.
_SC_BC_STRING_MAX	Specifies the maximum length of a string constant allowed by the **bc** command.
_SC_CHILD_MAX	Specifies the number of simultaneous processes per real user ID.

Item	Description
_SC_CLK_TCK	Indicates the clock-tick increment as defined by the **CLK_TCK** in the **time.h** file.
_SC_COLL_WEIGHTS_MAX	Specifies the maximum number of weights that can be assigned to an entry of the **LC_COLLATE** keyword in the locale definition file.
_SC_DELAYTIMER_MAX	Maximum number of Timer expiration overruns.
_SC_EXPR_NEST_MAX	Specifies the maximum number of expressions that can be nested within parentheses by the **expr** command.
_SC_JOB_CONTROL	If this symbol is defined, job control is supported.
_SC_IOV_MAX	Specifies the maximum number of iovec structures one process has available for use with the **readv** and **writev** subroutines.
_SC_LARGE_PAGESIZE	Size (in bytes) of a large-page.
_SC_LINE_MAX	Specifies the maximum byte length of a command's input line (either standard input or another file) when a command is described as processing text files. The length includes room for the trailing new-line character.
_SC_LOGIN_NAME_MAX	Maximum length of a login name.
_SC_MQ_OPEN_MAX	Maximum number of open message queue descriptors.
_SC_MQ_PRIO_MAX	Maximum number of message priorities.
_SC_MEMLOCK	Implementation supports the Process Memory Locking option.
_SC_MEMLOCK_RANGE	Implementation supports the Range Memory Locking option.
_SC_MEMORY_PROTECTION	Implementation supports the Memory Protection option.
_SC_MESSAGE_PASSING	Implementation supports the Message Passing option.
_SC_NGROUPS_MAX	Specifies the maximum number of simultaneous supplementary group IDs per process.
_SC_OPEN_MAX	Specifies the maximum number of files that one process can have open at any one time.
_SC_PASS_MAX	Specifies the maximum number of significant characters in a password (not including the terminating null character).
_SC_PASS_MAX	Maximum number of significant bytes in a password.
_SC_PAGESIZE	Equivalent to _SC_PAGE_SIZE.
_SC_PAGE_SIZE	Size in bytes of a page.
_SC_PRIORITIZED_IO	Implementation supports the Prioritized Input and Output option.
_SC_PRIORITY_SCHEDULING	Implementation supports the Process Scheduling option.
_SC_RE_DUP_MAX	Specifies the maximum number of repeated occurrences of a regular expression permitted when using the **\{** *m*, *n* **\}** interval notation.
_SC_RTSIG_MAX	Maximum number of Realtime Signals reserved for applications use.

Item	Description
_SC_REALTIME_SIGNALS	Implementation supports the Realtime Signals Extension option.
_SC_SAVED_IDS	If this symbol is defined, each process has a saved set-user ID and set-group ID.
_SC_SEM_NSEMS_MAX	Maximum number of Semaphores per process.

Item	Description
_SC_SEM_VALUE_MAX	Maximum value a Semaphore may have.
_SC_SEMAPHORES	Implementation supports the Semaphores option.
_SC_SHARED_MEMORY_OBJECTS	Implementation supports the Shared Memory Objects option.
_SC_SIGQUEUE_MAX	Maximum number of signals a process may send and have pending at any time.
_SC_STREAM_MAX	Specifies the maximum number of streams that one process can have open simultaneously.
_SC_SYNCHRONIZED_IO	Implementation supports the Synchronised Input and Output option.
_SC_TIMER_MAX	Maximum number of per-process Timers.
_SC_TIMERS	Implementation supports the Timers option.
_SC_TZNAME_MAX	Specifies the maximum number of bytes supported for the name of a time zone (not of the **TZ** value).
_SC_VERSION	Indicates that the version or revision number of the POSIX standard is implemented to indicate the 4-digit year and 2-digit month that the standard was approved by the IEEE Standards Board. This value is currently the long integer 198808.
_SC_XBS5_ILP32_OFF32	Implementation provides a C-language compilation environment with 32-bit int, long, pointer and off_t types.
_SC_XBS5_ILP32_OFFBIG	Implementation provides a C-language compilation environment with 32-bit int, long and pointer types and an off_t type using at least 64 bits.
_SC_XBS5_LP64_OFF64	Implementation provides a C-language compilation environment with 32-bit int and 64-bit long, pointer and off_t types.
_SC_XBS5_LPBIG_OFFBIG	Implementation provides a C-language compilation environment with an int type using at least 32 bits and long, pointer and off_t types using at least 64 bits.
_SC_XOPEN_CRYPT	Indicates that the system supports the X/Open Encryption Feature Group.
_SC_XOPEN_LEGACY	The implementation supports the Legacy Feature Group.
_SC_XOPEN_REALTIME	The implementation supports the X/Open Realtime Feature Group.
_SC_XOPEN_REALTIME_THREADS	The implementation supports the X/Open Realtime Threads Feature Group.
_SC_XOPEN_ENH_I18N	Indicates that the system supports the X/Open Enhanced Internationalization Feature Group.
_SC_XOPEN_SHM	Indicates that the system supports the X/Open Shared Memory Feature Group.
_SC_XOPEN_VERSION	Indicates that the version or revision number of the X/Open standard is implemented.
_SC_XOPEN_XCU_VERSION	Specifies the value describing the current version of the XCU specification.
_SC_ATEXIT_MAX	Specifies the maximum number of register functions for the **atexit** subroutine.
_SC_PAGE_SIZE	Specifies page-size granularity of memory.

Item	Description
_SC_AES_OS_VERSION	Indicates OSF AES version.
_SC_2_VERSION	Specifies the value describing the current version of POSIX.2.
_SC_2_C_BIND	Indicates that the system supports the C Language binding option.
_SC_2_C_CHAR_TERM	Indicates that the system supports at least one terminal type.
_SC_2_C_DEV	Indicates that the system supports the C Language Development Utilities Option.
_SC_2_C_VERSION	Specifies the value describing the current version of POSIX.2 with the C Language binding.
_SC_2_FORT_DEV	Indicates that the system supports the FORTRAN Development Utilities Option.
_SC_2_FORT_RUN	Indicates that the system supports the FORTRAN Development Utilities Option.
_SC_2_LOCALEDEF	Indicates that the system supports the creation of locales.
_SC_2_SW_DEV	Indicates that the system supports the Software Development Utilities Option.
_SC_2_UPE	Indicates that the system supports the User Portability Utilities Option.
_SC_NPROCESSORS_CONF	Number of processors configured.
_SC_NPROCESSORS_ONLN	Number of processors online.
_SC_THREAD_DATAKEYS_MAX	Maximum number of data keys that can be defined in a process.

Item	Description
_SC_THREAD_DESTRUCTOR_ITERATIONS	Maximum number attempts made to destroy a thread's thread-specific data.
_SC_THREAD_KEYS_MAX	Maximum number of data keys per process.
_SC_THREAD_STACK_MIN	Minimum value for the threads stack size.
_SC_THREAD_THREADS_MAX	Maximum number of threads within a process.
_SC_REENTRANT_FUNCTIONS	System supports reentrant functions (reentrant functions must be used in multi-threaded applications).
_SC_THREADS	System supports POSIX threads.
_SC_THREAD_ATTR_STACKADDR	System supports the stack address option for POSIX threads (stackaddr attribute of threads).
_SC_THREAD_ATTR_STACKSIZE	System supports the stack size option for POSIX threads (stacksize attribute of threads).
_SC_THREAD_PRIORITY_SCHEDULING	System supports the priority scheduling for POSIX threads.
_SC_THREAD_PRIO_INHERIT	System supports the priority inheritance protocol for POSIX threads (priority inversion protocol for mutexes).
_SC_THREAD_PRIO_PROTECT	System supports the priority ceiling protocol for POSIX threads (priority inversion protocol for mutexes).
_SC_THREAD_PROCESS_SHARED	System supports the process sharing option for POSIX threads (pshared attribute of mutexes and conditions).
_SC_TTY_NAME_MAX	Maximum length of a terminal device name.

Item	Description
_SC_SYNCHRONIZED_IO	Implementation supports the **Synchronized Input and Output** option.
_SC_FSYNC	Implementation supports the **File Synchronization** option.
_SC_MAPPED_FILES	Implementation supports the **Memory Mapped Files** option.
_SC_LPAR_ENABLED	Indicates whether LPARs are enabled or not.
_SC_AIX_KERNEL_BITMODE	Determines if the kernel is 32-bit or 64-bit.
_SC_AIX_REALMEM	Determines the amount of real memory in kilobytes.
_SC_AIX_HARDWARE_BITMODE	Determines whether the machine is 32-bit or 64-bit.
_SC_AIX_MP_CAPABLE	Determines if the hardware is MP-capable or not. **Note:** The **_SC_AIX_MP_CAPABLE** variable is available only to the root user.
_SC_AIX_UKEYS	Number of user-keys available. A value of 0 indicates that user-keys and the interfaces that manage them are not available.

Note: The **_SYNCHRONIZED_IO**, **_SC_FSYNC**, and **SC_MAPPED_FILES** commands apply to operating system version 4.3 and later releases.

The values returned for the variables supported by the system do not change during the lifetime of the process making the call.

Return Values

If the **sysconf** subroutine is successful, the current value of the system variable is returned. The returned value cannot be more restrictive than the corresponding value described to the application by the **limits.h**, **time.h**, or **unistd.h** file at compile time. The returned value does not change during the lifetime of the calling process. If the **sysconf** subroutine is unsuccessful, a value of -1 is returned.

Error Codes

If the *Name* parameter is invalid, a value of -1 is returned and the **errno** global variable is set to indicate the error. If the *Name* parameter is valid but is a variable not supported by the system, a value of -1 is returned, and the **errno** global variable is set to a value of **EINVAL**. If the system variable **_SC_AIX_MP_CAPABLE** is accessed by a non-root user, a value of -1 is returned and the **errno** global variable indicates the error

File

Item	Description
/usr/include/limits.h	Contains system-defined limits.

sysconfig Subroutine

Purpose

Provides a service for controlling system/kernel configuration.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/sysconfig.h>
```

```
int sysconfig ( Cmd,  Parmp,  Parmlen)
int Cmd;
void *Parmp;
int Parmlen;
```

Description

The **sysconfig** subroutine is used to customize the operating system. This subroutine provides a means of loading, unloading, and configuring kernel extensions. These kernel extensions can be additional kernel services, system calls, device drivers, or File systems in *Operating system and device management*. The **sysconfig** subroutine also provides the ability to read and set system run-time operating parameters.

Use of the **sysconfig** subroutine requires appropriate privilege.

The particular operation that the **sysconfig** subroutine provides is defined by the value of the *Cmd* parameter. The following operations are defined:

Item	Description
SYS_KLOAD ("SYS_KLOAD sysconfig Operation" on page 2115)	Loads a kernel extension object file into kernel memory.
SYS_SINGLELOAD ("SYS_SINGLELOAD sysconfig Operation" on page 2121)	Loads a kernel extension object file only if it is not already loaded.
SYS_QUERYLOAD ("SYS_QUERYLOAD sysconfig Operation" on page 2119)	Determines if a specified kernel object file is loaded.
SYS_KULOAD ("SYS_KULOAD sysconfig Operation" on page 2117)	Unloads a previously loaded kernel object file.
SYS_QDVSW ("SYS_QDVSW sysconfig Operation" on page 2118)	Checks the status of a device switch entry in the device switch table.
SYS_CFGDD ("SYS_CFGDD sysconfig Operation" on page 2111)	Calls the specified device driver configuration routine (module entry point).
SYS_CFGKMOD ("SYS_CFGKMOD sysconfig Operation" on page 2112)	Calls the specified module at its module entry point for configuration purposes.

Item	Description
SYS_GETPARMS (“SYS_GETPARMS sysconfig Operation” on page 2114)	Returns a structure containing the current values of run-time system parameters found in the **var** structure.
SYS_SETPARMS (“SYS_SETPARMS sysconfig Operation” on page 2120)	Sets run-time system parameters from a caller-provided structure.
SYS_GETLPARINFO (“SYS_GETLPAR_INFO sysconfig Operation” on page 2113)	Copies the system LPAR information into a user-allocated buffer.

In addition, the **SYS_64BIT** flag can be bitwise or'ed with the *Cmd* parameter (if the *Cmd* parameter is **SYS_KLOAD** or **SYS_SINGLELOAD**). For kernel extensions, this indicates that the kernel extension does not export 64-bit system calls, but that all 32-bit system calls also work for 64-bit applications. For device drivers, this indicates that the device driver can be used by 64-bit applications.

“Loader Symbol Binding Support” on page 2116 explains the symbol binding support provided when loading kernel object files.

Parameters

Item	Description
Cmd	Specifies the function that the **sysconfig** subroutine is to perform.
Parmp	Specifies a user-provided structure.
Parmlen	Specifies the length of the user-provided structure indicated by the *Parmp* parameter.

Return Values

These sysconfig operations return a value of 0 upon successful completion of the subroutine. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Any sysconfig operation requiring a structure from the caller fails if the structure is not entirely within memory addressable by the calling process. A return value of -1 is passed back and the **errno** global variable is set to **EFAULT**.

SYS_CFGDD sysconfig Operation

Purpose

Calls a previously loaded device driver at its module entry point.

Description

The **SYS_CFGDD** sysconfig operation calls a previously loaded device driver at its module entry point. The device driver's module entry point, by convention, is its **ddconfig** entry point. The **SYS_CFGDD** operation is typically invoked by device configure or unconfigure methods to initialize or terminate a device driver, or to request device vital product data.

The **sysconfig** subroutine puts no restrictions on the command code passed to the device driver. This allows the device driver's **ddconfig** entry point to provide additional services, if desired.

The *parmp* parameter on the **SYS_CFGDD** operation points to a **cfg_dd** structure defined in the **sys/sysconfig.h** file. The *parmlen* parameter on the **sysconfig** system call should be set to the size of this structure.

If the *kmid* variable in the **cfg_dd** structure is 0, the desired device driver is assumed to be already installed in the device switch table. The major portion of the device number (passed in the `devno` field in the **cfg_dd** structure) is used as an index into the device switch table. The device switch table entry indexed by this `devno` field contains the device driver's **ddconfig** entry point to be called.

If the *kmid* variable is not 0, it contains the module ID to use in calling the device driver. A **uio** structure is used to pass the address and length of the device-dependent structure, specified by the `cfg_dd.ddsptr` and `cfg_dd.ddslen` fields, to the device driver being called.

The **ddconfig** device driver entry point provides information on how to define the **ddconfig** subroutine.

The device driver to be called is responsible for using the appropriate routines to copy the device-dependent structure (DDS) from user to kernel space.

Return Values

If the **SYS_CFGDD** operation successfully calls the specified device driver, the return code from the **ddconfig** subroutine determines the value returned by this subroutine. If the **ddconfig** routine's return code is 0, then the value returned by the **sysconfig** subroutine is 0. Otherwise the value returned is a -1, and the **errno** global variable is set to the return code provided by the device driver **ddconfig** subroutine.

Error Codes

Errors detected by the **SYS_CFGDD** operation result in the following values for the **errno** global variable:

Item	Description
EACCES	The calling process does not have the required privilege.
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the system call. This error is also returned if an I/O error occurred when accessing data in this area.
EINVAL	Invalid module ID.
ENODEV	Module ID specified by the **cfg_dd.kmid** field was 0, and an invalid or undefined **devno** value was specified.

SYS_CFGKMOD sysconfig Operation

Purpose

Invokes a previously loaded kernel object file at its module entry point.

Description

The **SYS_CFGKMOD** sysconfig operation invokes a previously loaded kernel object file at its module entry point, typically for initialization or termination functions. The **SYS_CFGDD** ("SYS_CFGDD sysconfig Operation" on page 2111) operation performs a similar function for device drivers.

The *parmp* parameter on the **sysconfig** subroutine points to a **cfg_kmod** structure, which is defined in the **sys/sysconfig.h** file. The `kmid` field in this structure specifies the kernel module ID of the module to invoke. This value is returned when using the **SYS_KLOAD** ("SYS_KLOAD sysconfig Operation" on page 2115) or **SYS_SINGLELOAD** ("SYS_SINGLELOAD sysconfig Operation" on page 2121) operation to load the object file.

The cmd field in the **cfg_kmod** structure is a module-dependent parameter specifying the action that the routine at the module's entry point should perform. This is typically used for initialization and termination commands after loading and prior to unloading the object file.

The mdiptr field in the **cfg_kmod** structure points to a module-dependent structure whose size is specified by the mdilen field. This field is used to provide module-dependent information to the module to be called. If no such information is needed, the mdiptr field can be null.

If the mdiptr field is not null, then the **SYS_CFGKMOD** operation builds a **uio** structure describing the address and length of the module-dependent information in the caller's address space. The mdiptr and mdilen fields are used to fill in the fields of this **uio** structure. The module is then called at its module entry point with the *cmd* parameter and a pointer to the **uio** structure. If there is no module-dependent information to be provided, the *uiop* parameter passed to the module's entry point is set to null.

The module's entry point should be defined as follows:

```
int module_entry(cmd, uiop)
int cmd;
struct uio *uiop;
```

The definition of the module-dependent information and its length is specific to the module being configured. The called module is responsible for using the appropriate routines to copy the module-dependent information from user to kernel space.

Return Values

If the kernel module to be invoked is successfully called, its return code determines the value that is returned by the **SYS_CFGKMOD** operation. If the called module's return code is 0, then the value returned by the **sysconfig** subroutine is 0. Otherwise the value returned is -1 and the **errno** global variable is set to the called module's return code.

Error Codes

Errors detected by the **SYS_CFGKMOD** operation result in the following values for the **errno** global variable:

Item	Description
EINVAL	Invalid module ID.
EACCES	The calling process does not have the required privilege.
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the system call. This error is also returned if an I/O error occurred when accessing data in this area.

File

Item	Description
sys/sysconfig.h	Contains structure definitions.

SYS_GETLPAR_INFO sysconfig Operation

Purpose

Copies the system LPAR information into a user-allocated buffer.

Description

The **SYS_GETLPAR_INFO** sysconfig operation copies the system LPAR information into a user-allocated buffer.

The *parmp* parameter on the **sysconfig** subroutine points to a structure of type **getlpar_info**. Within the **getlpar_info** structure, the *lpar_namelen* field must be set by the user to the maximum length of the character buffer pointed to by *lpar_name*. On return, the *lpar_namelen* field will have its value replaced by the acual length of the *lpar_name* field. However, only the minimum of the actual length or the length provided by the user will be copied into the buffer pointed to by *lpar_name*. The *lpar_namesz*, *lpar_num*, and *lpar_name* fields will contain valid data on returning from the call only if the system is running as an LPAR as indicated by the value of the *lpar_flags* field being equal to **LPAR_ENABLED**.

If a value of 0 is specified for the *lpar_namesz* field, the partition name will not be copied out.

If the system is not an LPAR (namely it is running as an SMP system), but it is LPAR-capable, the **LPAR_CAPABLE** flag will be set on return.

The **getlpar_info** structure is defined below:

```
lpar_flags      unsigned short          LPAR_ENABLED: System is LPAR enabled.
                                        LPAR_CAPABLE: System is LPAR capable, but running in
SMP mode.
lpar_namesz     unsigned short          Size of partition name.
lpar_num        int                     Partition Number.
lpar_name       char *                  Partition Name.
```

Note: The *parmlen* parameter (which is the third parameter to the **sysconfig** system call) is ignored by the **SYS_GETLPAR_INFO** sysconfig operation.

Error Codes

The **SYS_GETLPAR_INFO** operation returns a value of -1 if an error occurs and the **errno** global variable is set to one of the following error codes:

Item	Description
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the subroutine or the *lpar_name* field in the **getlpar_info** structure. This error is also returned if an I/O error occurred when accessing data in any of these areas.
EINVAL	Invalid command parameter to the **sysconfig** subroutine.

Files

Item	Description
sys/ sysconfig.h	Contains structure definitions and flags.

SYS_GETPARMS sysconfig Operation

Purpose

Copies the system parameter structure into a user-specified buffer.

Description

The **SYS_GETPARMS** sysconfig operation copies the system parameter **var** structure into a user-allocated buffer. This structure may be used for informational purposes alone or prior to setting specific system parameters.

In order to set system parameters, the required fields in the **var** structure must be modified, and then the **SYS_SETPARMS** (“SYS_SETPARMS sysconfig Operation” on page 2120) operation can be called to change the system run-time operating parameters to the desired state.

The *parmp* parameter on the **sysconfig** subroutine points to a buffer that is to contain all or part of the **var** structure defined in the **sys/var.h** file. The fields in the **var_hdr** part of the **var** structure are used for parameter update control.

The *parmlen* parameter on the system call should be set to the length of the **var** structure or to the number of bytes of the structure that is desired. The complete definition of the system parameters structure can be found in the **sys/var.h** file.

Return Values

The **SYS_GETPARMS** operation returns a value of -1 if an error occurs and the **errno** global variable is set to one of the following error codes.

Error Codes

Item	Description
EACCES	The calling process does not have the required privilege.
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the subroutine. This error is also returned if an I/O error occurred when accessing data in this area.

File

Item	Description
sys/var.h	Contains structure definitions.

SYS_KLOAD sysconfig Operation

Purpose

Loads a kernel extension into the kernel.

Description

The **SYS_KLOAD sysconfig** operation is used to load a kernel extension object file specified by a path name into the kernel. A kernel module ID for that instance of the module is returned. The **SYS_KLOAD** operation loads a new copy of the object file into the kernel even though one or more copies of the specified object file may have already been loaded into the kernel. The returned module ID can then be used for any of these three functions:

- Subsequent invocation of the module's entry point (using the **SYS_CFGKMOD** (“SYS_CFGKMOD sysconfig Operation” on page 2112) operation)
- Invocation of a device driver's **ddconfig** subroutine (using the **SYS_CFGDD** (“SYS_CFGDD sysconfig Operation” on page 2111) operation)
- Unloading the kernel module (using the **SYS_KULOAD** (“SYS_KULOAD sysconfig Operation” on page 2117) operation).

The *parmp* parameter on the **sysconfig** subroutine must point to a **cfg_load** structure, (defined in the **sys/sysconfig.h** file), with the `path` field specifying the path name for a valid kernel object file. The *parmlen* parameter should be set to the size of the **cfg_load** structure.

Note: A separate **sysconfig** operation, the **SYS_SINGLELOAD** ("SYS_SINGLELOAD sysconfig Operation" on page 2121) operation, also loads kernel extensions. This operation, however, only loads the requested object file if not already loaded.

Loader Symbol Binding Support

The following information describes the symbol binding support provided when loading kernel object files.

Importing Symbols

Symbols imported from the kernel name space are resolved with symbols that exist in the corresponding kernel name space at the time of the load. (Symbols are imported from the kernel name space by specifying the `#!/unix` character string as the first field in an import list at link-edit time.)

Kernel modules can also import symbols from other kernel object files. These other kernel object files are loaded along with the specified object file if they are required to resolve the imported symbols.

Finding Directory Locations for Unqualified File Names

If the module header contains an unqualified base file name for the symbol (that is, no **/** [slash] characters in the name), a libpath search string is used to find the location of the shared object file required to resolve imported symbols. This libpath search string can be taken from one of two places. If the `libpath` field in the **cfg_load** structure is not null, then it points to a character string specifying the libpath to be used. However, if the `libpath` field is null, then the libpath is taken from the module header of the object file specified by the `path` field in the same (**cfg_load**) structure.

The libpath specification found in object files loaded in order to resolve imported symbols is not used.

The kernel loader service does not support deferred symbol resolution. The load of the kernel object file is terminated with an error if any imported symbols cannot be resolved.

Exporting Symbols

Any symbols exported by the specified kernel object file are added to the corresponding kernel name space. This makes these symbols available to other subsequently loaded kernel object files. Any symbols specified with the **SYSCALL** keyword in the export list at link-edit time are added to the system call table at load time. These symbols are then available to application programs as a system call. Symbols can be added to the 32-bit and 64-bit system call tables separately by using the **syscall32** and **syscall64** keywords. Symbols can be added to both system call tables by using the **syscall3264** keyword. A kernel extension that just exports 32-bit system calls can have all its system calls exported to 64-bit as well by passing the **SYS_64BIT** flag ORed with the **SYS_KLOAD** command to **sysconfig**.

Kernel object files loaded on behalf of the specified kernel object file to resolve imported symbols do not have their exported symbols added to the corresponding kernel name space.

These object files are considered private since they do not export symbols to the kernel name space. For these types of object files, a new copy of the object file is loaded on each **SYS_KLOAD** operation of a kernel extension that imports symbols from the private object file. In order for a kernel extension to add its exported symbols to the kernel name space, it must be explicitly loaded with the **SYS_KLOAD** operation before any other object files using the symbols are loaded. For kernel extensions of this type (those exporting symbols to the kernel name space), typically only one copy of the object file should ever be loaded.

Return Values

If the object file is loaded without error, the module ID is returned in the *kmid* variable within the **cfg_load** structure and the subroutine returns a value of **0**.

Error Codes

On error, the subroutine returns a value of **-1** and the **errno** global variable is set to one of the following values:

Item	Description
EACCES	One of the following reasons applies: • The calling process does not have the required privilege. • An object module to be loaded is not an ordinary file. • The mode of the object module file denies read-only permission.
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the system call. This error is also returned if an I/O error occurred when accessing data in this area.
ENOEXEC	The program file has the appropriate access permission, but has an invalid XCOFF object file indication in its header. The **SYS_KLOAD** operation only supports loading of XCOFF object files. This error is also returned if the loader is unable to resolve an imported symbol.
EINVAL	The program file has a valid XCOFF indicator in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.
ENOMEM	The load requires more kernel memory than is allowed by the system-imposed maximum.
ETXTBSY	The object file is currently open for writing by some process.

File

Item	Description
sys/sysconfig.h	Contains structure definitions.

SYS_KULOAD sysconfig Operation

Purpose

Unloads a loaded kernel object file and any imported kernel object files that were loaded with it.

Description

The **SYS_KULOAD** sysconfig operation unloads a previously loaded kernel file and any imported kernel object files that were automatically loaded with it. It does this by decrementing the load and use counts of the specified object file and any object file having symbols imported by the specified object file.

The *parmp* parameter on the **sysconfig** subroutine should point to a **cfg_load** structure, as described for the **SYS_KLOAD** ("SYS_KLOAD sysconfig Operation" on page 2115) operation. The `kmid` field should specify the kernel module ID that was returned when the object file was loaded by the **SYS_KLOAD** or **SYS_SINGLELOAD** ("SYS_SINGLELOAD sysconfig Operation" on page 2121) operation. The `path` and `libpath` fields are not used for this command and can be set to null. The *parmlen* parameter should be set to the size of the **cfg_load** structure.

Upon successful completion, the specified object file (and any other object files containing symbols that the specified object file imports) will have their load and use counts decremented. If there are no users of any of the module's exports and its load count is 0, then the object file is immediately unloaded.

However, if there are users of this module (that is, modules bound to this module's exported symbols), the specified module is not unloaded. Instead, it is unloaded on some subsequent unload request, when its use and load counts have gone to 0. The specified module is not in fact unloaded until all current users have been unloaded.

Note:

1. Care must be taken to ensure that a subroutine has freed all of its system resources before being unloaded. For example, a device driver is typically prepared for unloading by using the **SYS_CFGDD** ("SYS_CFGDD sysconfig Operation" on page 2111) operation and specifying termination.

2. If the use count is not 0, and you cannot force it to 0, the only way to terminate operation of the kernel extension is to reboot the machine.

"Loader Symbol Binding Support" on page 2116 explains the symbol binding support provided when loading kernel object files.

Return Values

If the unload operation is successful or the specified object file load count is successfully decremented, a value of 0 is returned.

Error Codes

On error, the specified file and any imported files are not unloaded, nor are their load and use counts decremented. A value of -1 is returned and the **errno** global variable is set to one of the following:

Item	Description
EACCES	The calling process does not have the required privilege.
EINVAL	Invalid module ID or the specified module is no longer loaded or already has a load count of 0.
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided to the subroutine. This error is also returned if an I/O error occurred when accessing data in this area.

SYS_QDVSW sysconfig Operation

Purpose

Checks the status of a device switch entry in the device switch table.

Description

The **SYS_QDVSW** sysconfig operation checks the status of a device switch entry in the device switch table.

The *parmp* parameter on the **sysconfig** subroutine points to a **qry_devsw** structure defined in the **sys/sysconfig.h** file. The *parmlen* parameter on the subroutine should be set to the length of the **qry_devsw** structure.

The `qry_devsw` field in the **qry_devsw** structure is modified to reflect the status of the device switch entry specified by the `qry_devsw` field. (Only the major portion of the `devno` field is relevant.) The following flags can be returned in the `status` field:

Item	Description
DSW_UNDEFINED	The device switch entry is not defined if this flag has a value of 0 on return.
DSW_DEFINED	The device switch entry is defined.
DSW_CREAD	The device driver in this device switch entry provides a routine for character reads or raw input. This flag is set when the device driver provides a **ddread** entry point.
DSW_CWRITE	The device driver in this device switch entry provides a routine for character writes or raw output. This flag is set when the device driver provides a **ddwrite** entry point.
DSW_BLOCK	The device switch entry is defined by a block device driver. This flag is set when the device driver provides a **ddstrategy** entry point.
DSW_MPX	The device switch entry is defined by a multiplexed device driver. This flag is set when the device driver provides a **ddmpx** entry point.

Item	Description
DSW_SELECT	The device driver in this device switch entry provides a routine for handling the **select** (“select Subroutine” on page 1859) or **poll** subroutines. This flag is set when the device driver provides a **ddselect** entry point.
DSW_DUMP	The device driver defined by this device switch entry provides the capability to support one or more of its devices as targets for a kernel dump. This flag is set when the device driver has provided a **dddump** entry point.
DSW_CONSOLE	The device switch entry is defined by the console device driver.
DSW_TCPATH	The device driver in this device switch entry supports devices that are considered to be in the trusted computing path and provides support for the **revoke** (“revoke Subroutine” on page 1749) and **frevoke** subroutines. This flag is set when the device driver provides a **ddrevoke** entry point.
DSW_OPENED	The device switch entry is defined and the device has outstanding opens. This flag is set when the device driver has at least one outstanding open.

The **DSW_UNDEFINED** condition is indicated when the device switch entry has not been defined or has been defined and subsequently deleted. Multiple status flags may be set for other conditions of the device switch entry.

Return Values

If no error is detected, this operation returns with a value of 0. If an error is detected, the return value is set to a value of -1.

Error Codes

When an error is dected, the **errno** global variable is also set to one of the following values:

Item	Description
EACCES	The calling process does not have the required privilege.
EINVAL	Device number exceeds the maximum allowed by the kernel.
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the system call. This error is also returned if an I/O error occurred when accessing data in this area.

File

Item	Description
sys/sysconfig.h	Contains structure definitions.

SYS_QUERYLOAD sysconfig Operation

Purpose

Determines if a kernel object file has already been loaded.

Description

The **SYS_QUERYLOAD** sysconfig operation performs a query operation to determine if a given object file has been loaded. This object file is specified by the `path` field in the **cfg_load** structure passed in with the *parmp* parameter. This operation utilizes the same **cfg_load** structure that is specified for the **SYS_KLOAD** (“SYS_KLOAD sysconfig Operation” on page 2115) operation.

If the specified object file is not loaded, the kmid field in the **cfg_load** structure is set to a value of 0 on return. Otherwise, the kernel module ID of the module is returned in the kmid field. If multiple instances of the module have been loaded into the kernel, the module ID of the one most recently loaded is returned.

The libpath field in the **cfg_load** structure is not used for this option.

Note: A path-name comparison is done to determine if the specified object file has been loaded. However, this operation will erroneously return a *not loaded* condition if the path name to the object file is expressed differently than it was on a previous load request.

"Loader Symbol Binding Support" on page 2116 explains the symbol binding support provided when loading kernel object files.

Return Values

If the specified object file is found, the module ID is returned in the *kmid* variable within the **cfg_load** structure and the subroutine returns a 0. If the specified file is not found, a *kmid* variable of 0 is returned with a return code of 0.

Error Codes

On error, the subroutine returns a -1 and the **errno** global variable is set to one of the following values:

Item	Description
EACCES	The calling process does not have the required privilege.
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided on the subroutine. This error is also returned if an I/O error occurred when accessing data in this area.
EFAULT	The *path* parameter points to a location outside of the allocated address space of the process.
EIO	An I/O error occurred during the operation.

SYS_SETPARMS sysconfig Operation

Purpose

Sets the kernel run-time tunable parameters.

Description

The **SYS_SETPARMS** sysconfig operation sets the current system parameters from a copy of the system parameter **var** structure provided by the caller. Only the run-time tunable parameters in the **var** structure can be set by this subroutine.

If the var_vers and var_gen values in the caller-provided structure do not match the var_vers and var_gen values in the current system **var** structure, no parameters are modified and an error is returned. The var_vers, var_gen, and var_size fields in the structure should not be altered. The var_vers value is assigned by the kernel and is used to insure that the correct version of the structure is being used. The **var_gen** value is a generation number having a new value for each read of the structure. This provides consistency between the data read by the **SYS_GETPARMS** ("SYS_GETPARMS sysconfig Operation" on page 2114) operation and the data written by the **SYS_SETPARMS** operation.

The *parmp* parameter on the **sysconfig** subroutine points to a buffer that contains all or part of the **var** structure as defined in the **sys/var.h** file.

The *parmlen* parameter on the subroutine should be set either to the length of the **var** structure or to the size of the structure containing the parameters to be modified. The number of system parameters

modified by this operation is determined either by the *parmlen* parameter value or by the `var_size` field in the caller-provided **var** structure. (The smaller of the two values is used.)

The structure provided by the caller must contain at least the header fields of the **var** structure. Otherwise, an error will be returned. Partial modification of a parameter in the **var** structure can occur if the caller's data area does not contain enough data to end on a field boundary. It is up to the caller to ensure that this does not happen.

Return Values

The **SYS_SETPARMS** sysconfig operation returns a value of -1 if an error occurred.

Error Codes

When an error occurs, the **errno** global variable is set to one of the following values:

Item	Description
EACCES	The calling process does not have the required privilege.
EINVAL	One of the following error situations exists: • The `var_vers` version number of the provided structure does not match the version number of the current **var** structure. • The structure provided by the caller does not contain enough data to specify the header fields within the **var** structure. • One of the specified variable values is invalid or not allowed. On the return from the subroutine, the `var_vers` field in the caller-provided buffer contains the byte offset of the first variable in the structure that was detected in error.
EAGAIN	The `var_gen` generation number in the structure provided does not match the current generation number in the kernel. This occurs if consistency is lost between reads and writes of this structure. The caller should repeat the read, modify, and write operations on the structure.
EFAULT	The calling process does not have sufficient authority to access the data area described by the *parmp* and *parmlen* parameters provided to the subroutine. This error is also returned if an I/O error occurred when accessing data in this area.

File

Item	Description
sys/var.h	Contains structure definitions.

SYS_SINGLELOAD sysconfig Operation

Purpose

Loads a kernel extension module if it is not already loaded.

Description

The **SYS_SINGLELOAD** sysconfig operation is identical to the **SYS_KLOAD** ("SYS_KLOAD sysconfig Operation" on page 2115) operation, except that the **SYS_SINGLELOAD** operation loads the object file only if an object file with the same path name has not already been loaded into the corresponding kernel environment.

If an object file with the same path name has already been loaded, the module ID for that object file is returned in the kmid field and its load count incremented. If the object file is not loaded, this operation performs the load request exactly as defined for the **SYS_KLOAD** operation.

This option is useful in supporting global kernel routines where only one copy of the routine and its data can be present. Typically routines that export symbols to be added to the kernel name space are of this type.

Note: A path name comparison is done to determine if the same object file has already been loaded. However, this function will erroneously load a new copy of the object file into the kernel if the path name to the object file is expressed differently than it was on a previous load request.

"Loader Symbol Binding Support" on page 2116 explains the symbol binding support provided when loading kernel object files.

Return Values

The **SYS_SINGLELOAD** operation returns the same set of error codes that the **SYS_KLOAD** operation returns.

syslog, openlog, closelog, or setlogmask Subroutine

Purpose

Controls the system log.

Library

Standard C Library (**libc.a**)

Syntax

#include <syslog.h>

void openlog (*ID*, *LogOption*, *Facility***) const char** **ID*; **int** *LogOption*, *Facility*;

void syslog (*Priority*, *Value*,... **) int** *Priority*; **const char** **Value*;

void closelog ( )

int setlogmask(*MaskPriority***) int** *MaskPriority*;

void bsdlog (*Priority*, *Value*,...**) int** *Priority*; **const char** **Value*;

Description

Attention: Do not use the **syslog**, **openlog**, **closelog**, or **setlogmask** subroutine in a multithreaded environment. See the multithread alternatives in the **syslog_r** ("syslog_r, openlog_r, closelog_r, or setlogmask_r Subroutine" on page 2125), **openlog_r**, **closelog_r**, or **setlogmask_r** subroutine article. The **syslog** subroutine is not threadsafe; for threadsafe programs the **syslog_r** subroutine should be used instead.

The **syslog** subroutine writes messages onto the system log maintained by the **syslogd** command.

Note: Messages passed to **syslog** that are longer than 900 bytes may be truncated by **syslogd** before being logged.

The message is similar to the **printf** *fmt* string, with the difference that *%m* is replaced by the current error message obtained from the **errno** global variable. A trailing new-line can be added to the message if needed.

Messages are read by the **syslogd** command and written to the system console or log file, or forwarded to the **syslogd** command on the appropriate host.

If special processing is required, the **openlog** subroutine can be used to initialize the log file.

Messages are tagged with codes indicating the type of *Priority* for each. A *Priority* is encoded as a *Facility*, which describes the part of the system generating the message, and as a level, which indicates the severity of the message.

If the **syslog** subroutine cannot pass the message to the **syslogd** command, it writes the message on the **/dev/console** file, provided the **LOG_CONS** option is set.

The **closelog** subroutine closes the log file.

The **setlogmask** subroutine uses the bit mask in the *MaskPriority* parameter to set the new log priority mask and returns the previous mask.

The **LOG_MASK** and **LOG_UPTO** macros in the **sys/syslog.h** file are used to create the priority mask. Calls to the **syslog** subroutine with a priority mask that does not allow logging of that particular level of message causes the subroutine to return without logging the message.

Parameters

Item	Description
ID	Contains a string that is attached to the beginning of every message. The *Facility* parameter encodes a default facility from the previous list to be assigned to messages that do not have an explicit facility encoded.
LogOption	Specifies a bit field that indicates logging options. The values of *LogOption* are: **LOG_CONS** Sends messages to the console if unable to send them to the **syslogd** command. This option is useful in daemon processes that have no controlling terminal. **LOG_NDELAY** Opens the connection to the **syslogd** command immediately, instead of when the first message is logged. This option is useful for programs that need to manage the order in which file descriptors are allocated. **LOG_NOWAIT** Logs messages to the console without waiting for forked children. Use this option for processes that enable notification of child termination through **SIGCHLD**; otherwise, the **syslog** subroutine may block, waiting for a child process whose exit status has already been collected. **LOG_ODELAY** Delays opening until the **syslog** subroutine is called. **LOG_PID** Logs the process ID with each message. This option is useful for identifying daemons.

Item	Description
Facility	Specifies which of the following values generated the message: **LOG_AUTH** Indicates the security authorization system: the **login** command, the **su** command, and so on. **LOG_DAEMON** Logs system daemons. **LOG_KERN** Logs messages generated by the kernel. Kernel processes should use the **bsdlog** routine to generate **syslog** messages. The syntax of **bsdlog** is identical to **syslog**. The **bsdlog** messages can only be created by kernel processes and must be of **LOG_KERN** priority. The **syslog** subroutine cannot log **LOG_KERN** facility messages. Instead it will log **LOG_USER** facility messages. **LOG_LPR** Logs the line printer spooling system. **LOG_LOCAL0 through LOG_LOCAL7** Reserved for local use. **LOG_MAIL** Logs the mail system. **LOG_NEWS** Logs the news subsystem. **LOG_UUCP** Logs the UUCP subsystem. **LOG_USER** Logs messages generated by user processes. This is the default facility when none is specified.
Priority	Specifies the part of the system generating the message, and as a level, indicates the severity of the message. The level of severity is selected from the following list: **LOG_ALERT** Indicates a condition that should be corrected immediately; for example, a corrupted database. **LOG_CRIT** Indicates critical conditions; for example, hard device errors. **LOG_DEBUG** Displays messages containing information useful to debug a program. **LOG_EMERG** Indicates a panic condition reported to all users; system is unusable. **LOG_ERR** Indicated error conditions. **LOG_INFO** Indicates general information messages. **LOG_NOTICE** Indicates a condition requiring special handling, but not an error condition. **LOG_WARNING** Logs warning messages.
MaskPriority	Enables logging for the levels indicated by the bits in the mask that are set and disabled where the bits are not set. The default mask allows all priorities to be logged.

Item	Description
Value	Specifies the values given in the *Value* parameters and follows the the same syntax as the **printf** subroutine *Format* parameter.

Examples

1. To log an error message concerning a possible security breach, such as the following, enter:

```
syslog (LOG_ALERT, "who:internal error 23");
```

2. To initialize the log file, set the log priority mask, and log an error message, enter:

```
openlog ("ftpd", LOG_PID, LOG_DAEMON);
setlogmask (LOG_UPTO (LOG_ERR));
syslog (LOG_INFO, "");
```

3. To log an error message from the system, enter:

```
syslog (LOG_INFO | LOG_LOCAL2, "foobar error: %m");
```

syslog_r, openlog_r, closelog_r, or setlogmask_r Subroutine

Purpose

Controls the system log.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <syslog.h>

int syslog_r (Priority, SysLogData, Format, . . .)
int  Priority;
struct syslog_data * SysLogData;
const char * Format;

int openlog_r (ID, LogOption, Facility, SysLogData)
const char * ID;
int LogOption;
int  Facility;

struct syslog_data *SysLogData;
void closelog_r (SysLogData)
struct syslog_data *SysLogData;

int setlogmask_r ( MaskPriority, SysLogData)
int MaskPriority;
struct syslog_data *SysLogData;
```

Description

The **syslog_r** subroutine writes messages onto the system log maintained by the **syslogd** daemon.

The messages are similar to the *Format* parameter in the **printf** subroutine, except that the %m field is replaced by the current error message obtained from the **errno** global variable. A trailing new-line character can be added to the message if needed.

Messages are read by the **syslogd** daemon and written to the system console or log file, or forwarded to the **syslogd** daemon on the appropriate host.

If a program requires special processing, you can use the **openlog_r** subroutine to initialize the log file.

The **syslog_r** subroutine takes as a second parameter a variable of the type **struct syslog_data**, which should be provided by the caller. When that variable is declared, it should be set to the **SYSLOG_DATA_INIT** value, which specifies an initialization macro defined in the **sys/syslog.h** file. Without initialization, the data structure used to support the thread safety is not set up and the **syslog_r** subroutine does not work properly.

Messages are tagged with codes indicating the type of *Priority* for each. A *Priority* is encoded as a *Facility*, which describes the part of the system generating the message, and as a level, which indicates the severity of the message.

If the **syslog_r** subroutine cannot pass the message to the **syslogd** daemon, it writes the message the **/dev/console** file, provided the **LOG_CONS** option is set.

The **closelog_r** subroutine closes the log file.

The **setlogmask_r** subroutine uses the bit mask in the *MaskPriority* parameter to set the new log priority mask and returns the previous mask.

The **LOG_MASK** and **LOG_UPTO** macros in the **sys/syslog.h** file are used to create the priority mask. Calls to the **syslog_r** subroutine with a priority mask that does not allow logging of that particular level of message causes the subroutine to return without logging the message.

Programs using this subroutine must link to the **libpthreads.a** library.

Parameters

Item	Description
Priority	Specifies the part of the system generating the message and indicates the level of severity of the message. The level of severity is selected from the following list: • A condition that should be corrected immediately, such as a corrupted database. • A critical condition, such as hard device errors. • A message containing information useful to debug a program. • A panic condition reported to all users, such as an unusable system. • An error condition. • A general information message. • A condition requiring special handling, other than an error condition. • A warning message.
SysLogData	Specifies a structure that contains the following information: • The file descriptor for the log file. • The status bits for the log file. • A string for tagging the log entry. • The mask of priorities to be logged. • The default facility code. • The address of the local logger.
Format	Specifies the format, given in the same format as for the **printf** subroutine.

Item	Description
ID	Contains a string attached to the beginning of every message. The Facility parameter encodes a default facility from the previous list to be assigned to messages that do not have an explicit facility encoded.
LogOption	Specifies a bit field that indicates logging options. The values of *LogOption* are: **LOG_CONS** Sends messages to the console if unable to send them to the **syslogd** command. This option is useful in daemon processes that have no controlling terminal. **LOG_NDELAY** Opens the connection to the **syslogd** command immediately, instead of when the first message is logged. This option is useful for programs that need to manage the order in which file descriptors are allocated. **LOG_NOWAIT** Logs messages to the console without waiting for forked children. Use this option for processes that enable notification of child termination through **SIGCHLD**; otherwise, the **syslog** subroutine may block, waiting for a child process whose exit status has already been collected. **LOG_ODELAY** Delays opening until the **syslog** subroutine is called. **LOG_PID** Logs the process ID with each message. This option is useful for identifying daemons.

Item	Description
Facility	Specifies which of the following values generated the message: **LOG_AUTH** Indicates the security authorization system: the **login** command, the **su** command, and so on. **LOG_DAEMON** Logs system daemons. **LOG_KERN** Logs messages generated by the kernel. Kernel processes should use the **bsdlog** routine to generate **syslog** messages. The syntax of **bsdlog** is identical to **syslog**. The **bsdlog** messages can only be created by kernel processes and must be of **LOG_KERN** priority. **LOG_LPR** Logs the line printer spooling system. **LOG_LOCAL0 through LOG_LOCAL7** Reserved for local use. **LOG_MAIL** Logs the mail system. **LOG_NEWS** Logs the news subsystem. **LOG_UUCP** Logs the UUCP subsystem. **LOG_USER** Logs messages generated by user processes. This is the default facility when none is specified. • Remote file systems, such as the Andrew File System. • The UUCP subsystem. • Messages generated by user processes. This is the default facility when none is specified.
MaskPriority	Enables logging for the levels indicated by the bits in the mask that are set, and disables logging where the bits are not set. The default mask allows all priorities to be logged.

Return Values

Item	Description
0	Indicates that the subroutine was successful.
-1	Indicates that the subroutine was not successful. Moves an error code, indicating the specific error, into the `errno` global variable.

Error Codes

When the **syslog_r** subroutine is unsuccessful, the `errno` global variable can be set to the following values:

Item	Description
EAGAIN	Exceeds the limit on the total number of processes running either system-wide or by a single user, or the system does not have the resources necessary to create another process.

Item	Description
EBADF	The syslogd daemon is not active.
ECONNRESET	The syslogd daemon stopped during the operation.
ENOBUFS	Buffer resources were not available.
ENOMEM	Not enough space exists for this process.
ENOTCONN	The syslogd daemon stopped during the operation.
EPROCLIM	If WLM is running, the limit on the number of processes or threads in the class might have been met.
EINVAL	The Priority parameter is not a valid parameter.

Examples

1. To log an error message concerning a possible security breach, enter:

```
syslog_r (LOG_ALERT, syslog_data_struct, "%s", "who:internal error 23");
```

2. To initialize the log file, set the log priority mask, and log an error message, enter:

```
openlog_r ("ftpd", LOG_PID, LOG_DAEMON, syslog_data_struct);
setlogmask_r (LOG_UPTO (LOG_ERR), syslog_data_struct);
syslog_r (LOG_INFO, syslog_data_struct, "");
```

3. To log an error message from the system, enter:

```
syslog_r (LOG_INFO | LOG_LOCAL2, syslog_data_struct, "system error: %m");
```

sys_parm Subroutine

Purpose

Provides a service for examining or setting kernel run-time tunable parameters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/var.h>

int sys_parm ( cmd,  parmflag,  parmp)
int cmd;
int parmflag;
struct vario *parmp;
```

Description

The **sys_parm** subroutine is used to query and/or customize run-time operating system parameters.

Note: This is a replacement service for **sysconfig** with respect to querying or changing information in the **var** structure. The **audit** subroutine or command can be used to audit changes to the **var** structure.

The **sys_parm** subroutine:

- Works on both 32 bit and 64 bit platforms

- Requires appropriate privilege for its use.

The following operations are supported:

Item	Description
SYSP_GET	Returns a structure containing the current value of the specified run-time parameter found in the **var** structure.
SYSP_SET	Sets the value of the specfied run-time parameter.

The run-time parameters that can be returned or set are found in the var structure as defined in var.h

Parameters

Item	Description
cmd	Specifies the **SYSP_GET** or **SYSP_SET** function.
parmflag	Specifies the parameter upon which the function will act.
parmp	Points to the user specified structure from which or to which the system parameter value is copied. *parmp* points to a structure of type **vario** as defined in var.h.

The **vario** structure is an abstraction of the various fields in the **var** structure for which each field is size invariant. The size of the data does not depend on the execution environment of the kernel being 32 or 64 bit or the calling application being 32 or 64 bit.

Examples

1. To examine the value of v.v_iostrun (collect disk usage statistics).

```
#include <sys/var.h>
#include <stdio.h>
struct vario myvar;
rc=sys_parm(SYSP_GET,SYSP_V_IOSTRUN,&myvar);
if(rc==0)
        printf("v.v_iostrun is set to %d\n",myvar.v.v_iostrun.value);
```

2. To change the value of v.v_iostrun (collect disk usage statistics).

```
#include <sys/var.h>
#include <stdio.h>
struct vario myvar;
myvar.v.v_iostrun.value=0; /* initialize to false */
rc=sys_parm(SYSP_SET,SYSP_V_IOSTRUN,&myvar);
if(rc==0)
        printf("disk usage statistics are not being collected\n");
```

Other parameters may be examined or set by changing the parmflag parameter.

Return Values

These operations return a value of 0 upon succesful completion of the subroutine. Otherwise or a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

Item	Description
EACCES	The calling process does not have the required privilege.
EINVAL	One of the following is true: • The command is neither **SYSP_GET** nor **SYSP_SET** • *parmflag* is out of range of parameters defined in var.h • The value specified in the *parmp* parameter is not a valid value for the field indicated by the *parmflag* parameter.
EFAULT	An invalid address was specified by the *parmp* parameter.

File

Item	Description
sys/var.h	Contains structure definitions.

system Subroutine

Purpose

Runs a shell command.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

int system ( String)
const char *String;
```

Description

The **system** subroutine passes the *String* parameter to the **sh** command as input. Then the **sh** command interprets the *String* parameter as a command and runs it.

The **system** subroutine calls the **fork** subroutine to create a child process that in turn uses the **exec l** subroutine to run the /**usr**/**bin**/**sh** command, which interprets the shell command contained in the *String* parameter. When invoked on the Trusted Path, the **system** subroutine runs the Trusted Path shell (/**usr**/**bin**/**tsh**). The current process waits until the shell has completed, then returns the exit status of the shell. The exit status of the shell is returned in the same manner as a call to the **wait** or **waitpid** subroutine, using the structures in the **sys/wait.h** file.

The **system** subroutine ignores the **SIGINT** and **SIGQUIT** signals, and blocks the **SIGCHILD** signal while waiting for the command specified by the *String* parameter to terminate. If this might cause the application to miss a signal that would have killed it, the application should use the value returned by the **system** subroutine to take the appropriate action if the command terminated due to receipt of a signal.

The **system** subroutine does not affect the termination status of any child of the calling process unless that process was created by the **system** subroutine. The **system** subroutine does not return until the child process has terminated.

Parameters

Item	Description
String	Specifies a valid **sh** shell command.

Note: The **system** subroutine runs only **sh** shell commands. The results are unpredictable if the *String* parameter is not a valid **sh** shell command.

Return Values

Upon successful completion, the **system** subroutine returns the exit status of the shell. The exit status of the shell is returned in the same manner as a call to the **wait** or **waitpid** subroutine, using the structures in the **sys/wait.h** file.

If the *String* parameter is a null pointer and a command processor is available, the **system** subroutine returns a nonzero value. If the **fork** subroutine fails or if the exit status of the shell cannot be obtained, the **system** subroutine returns a value of -1. If the **exec l** subroutine fails, the system subroutine returns a value of 127. In all cases, the **errno** global variable is set to indicate the error.

Error Codes

The **system** subroutine fails if any of the following are true:

Item	Description
EAGAIN	The system-imposed limit on the total number of running processes, either systemwide or by a single user ID, was exceeded.
EINTR	The **system** subroutine was interrupted by a signal that was caught before the requested process was started. The **EINTR** error code will never be returned after the requested process has begun.
ENOMEM	Insufficient storage space is available.

t

The following Base Operating System (BOS) runtime services begin with the letter *t*.

tan, tanf, tanl, tand32, tand64, and tand128 Subroutines

Purpose

Computes the tangent.

Syntax

```
#include <math.h>

float tanf (x)
float x;

long double tanl (x)
long double x;

double tan (x)
double x;
_Decimal32 tand32 (x)
_Decimal32 x;

_Decimal64 tand64 (x)
_Decimal64 x;

_Decimal128 tand128 (x)
_Decimal128 x;
```

Description

The **tan**, **tanf**, **tanl**, **tand32**, **tand64**, and **tand128** subroutines compute the tangent of the *x* parameter, measured in radians.

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these functions. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **tan**, **tanf**, **tanl**, **tand32**, **tand64**, and **tand128** subroutines return the tangent of *x*.

If the correct value would cause underflow, and is not representable, a range error may occur, and 0.0 is returned.

If *x* is NaN, a NaN is returned.

If *x* is ±0, *x* is returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

If *x* is ±Inf, a domain error occurs, and a NaN returned.

If the correct value would cause underflow, and is representable, a range error may occur and the correct value is returned.

If the correct value would cause overflow, a range error occurs and the **tan**, **tanf**, **tanl**, **tand32**, **tand64**, and **tand128** subroutines return the value of the macro **HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, and **HUGE_VAL_D128** respectively.

Error Codes

The **tan**, **tanf**, and **tanl** subroutines lose accuracy when passed a large value for the *x* parameter. Since the machine value of `pi` can only approximate its infinitely precise value, the remainder of x/(2 * `pi`) becomes less accurate as *x* becomes larger. Similar loss of accuracy occurs for the **tan**, **tanf**, and **tanl** subroutines during argument reduction of large arguments.

tanh, tanhf, tanhl, tanhd32, tanhd64, and tanhd128 Subroutines

The **tanhf**, **tanhl**, **tanh**, **tanhd32**, **tanhd64**, and **tanhd128** subroutines compute the hyperbolic tangent of the *x* .

An application wishing to check for error situations should set the **errno** global variable to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Purpose

Computes the hyperbolic tangent.

Syntax

```
#include <math.h>

float tanhf (x)
float x;
long double tanhl (x)
long double x;

double tanh (x)
double x;
_Decimal32 tanhd32 (x)
_Decimal32 x;

_Decimal64 tanhd64 (x)
_Decimal64 x;

_Decimal128 tanhd128 (x)
_Decimal128 x;
```

Description

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **tanhf**, **tanhl**, **tanh**, **tanhd32**, **tanhd64**, and **tanhd128** subroutines return the hyperbolic tangent of *x*.

If *x* is NaN, a NaN is returned.

If *x* is ±0, *x* is returned.

If *x* is ±Inf, ±1 is returned.

If *x* is subnormal, a range error may occur and *x* should be returned.

tcb Subroutine

Purpose

Alters the Trusted Computing Base (TCB) status of a file.

Library

Security Library (**libc.a**)

Syntax

```
#include <sys/tcb.h>

int tcb ( Path,  Flag)
char *Path;
int Flag;
```

Description

The **tcb** subroutine provides a mechanism to query or set the TCB attributes of a file.

This subroutine is not safe for use with multiple threads. To call this subroutine from a threaded application, enclose the call with the **_libs_rmutex** lock. See "Making a Subroutine Safe for Multiple Threads" in *General Programming Concepts: Writing and Debugging Programs* for more information about this lock.

Parameters

Item	Description
Path	Specifies the path name of the file whose TCB status is to be changed.
Flag	Specifies the function to be performed. Valid values are defined in the **sys/tcb.h** file and include the following: **TCB_ON** Enables the TCB attribute of a file. **TCB_OFF** Disables the Trusted Process and TCB attributes of a file. **TCB_QUERY** Queries the TCB status of a file. This function returns one of the preceding values.

Return Values

Upon successful completion, the **tcb** subroutine returns a value of 0 if the *Flags* parameter is either **TCB_ON** or **TCB_OFF**. If the *Flags* parameter is **TCB_QUERY**, the current status is returned. If the **tcb** subroutine fails, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **tcb** subroutine fails if one of the following is true:

Item	Description
EINVAL	The *Flags* parameter is not one of **TCB_ON**, **TCB_OFF**, or **TCB_QUERY**.
EPERM	Not authorized to perform this operation.
ENOENT	The file specified by the *Path* parameter does not exist.
EROFS	The file system is read-only.
EBUSY	The file specified by the *Path* parameter is currently open for writing.
EACCES	Access permission is denied for the file specified by the *Path* parameter.

Security

Access Control: The calling process must have search permission for the object named by the *Path* parameter. Only the root user can set the **tcb** attributes of a file.

tcdrain Subroutine

Purpose

Waits for output to complete.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <termios.h>

int tcdrain( FileDescriptor)
int FileDescriptor;
```

Description

The **tcdrain** subroutine waits until all output written to the object referred to by the *FileDescriptor* parameter has been transmitted.

Parameter

Item	Description
FileDescriptor	Specifies an open file descriptor.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **tcdrain** subroutine is unsuccessful if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.

Item	Description
EINTR	A signal interrupted the **tcdrain** subroutine.
EIO	The process group of the writing process is orphaned, and the writing process does not ignore or block the **SIGTTOU** signal.
ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Example

To wait until all output has been transmitted, enter:

```
rc = tcdrain(stdout);
```

tcflow Subroutine

Purpose

Performs flow control functions.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <termios.h>
```

```
int tcflow( FileDescriptor,  Action)
int FileDescriptor;
int Action;
```

Description

The **tcflow** subroutine suspends transmission or reception of data on the object referred to by the *FileDescriptor* parameter, depending on the value of the *Action* parameter.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.

Item	Description
Action	Specifies one of the following: **TCOOFF** Suspend output. **TCOON** Restart suspended output. **TCIOFF** Transmit a STOP character, which is intended to cause the terminal device to stop transmitting data to the system. See the description of IXOFF in the Input Modes section of the **termios.h** file. **TCION** Transmit a START character, which is intended to cause the terminal device to start transmitting data to the system. See the description of IXOFF in the Input Modes section of the **termios.h** file.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **tcflow** subroutine is unsuccessful if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.
EINVAL	The *Action* parameter does not specify a proper value.
EIO	The process group of the writing process is orphaned, and the writing process does not ignore or block the **SIGTTOU** signal.
ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Example

To restart output from a terminal device, enter:

```
rc = tcflow(stdout, TCION);
```

tcflush Subroutine

Purpose

Discards data from the specified queue.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <termios.h>
```

```
int tcflush( FileDescriptor,  QueueSelector)
int FileDescriptor;
int QueueSelector;
```

Description

The **tcflush** subroutine discards any data written to the object referred to by the *FileDescriptor* parameter, or data received but not read by the object referred to by *FileDescriptor*, depending on the value of the *QueueSelector* parameter.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.
QueueSelector	Specifies one of the following: **TCIFLUSH** Flush data received but not read. **TCOFLUSH** Flush data written but not transmitted. **TCIOFLUSH** Flush both of the following: • Data received but not read • Data written but not transmitted

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **tcflush** subroutine is unsuccessful if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.
EINVAL	The *QueueSelector* parameter does not specify a proper value.
EIO	The process group of the writing process is orphaned, and the writing process does not ignore or block the **SIGTTOU** signal.
ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Example

To flush the output queue, enter:

```
rc = tcflush(2, TCOFLUSH);
```

tcgetattr Subroutine

Purpose

Gets terminal state.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <termios.h>
```

```
int tcgetattr ( FileDescriptor,  TermiosPointer)
int FileDescriptor;
struct termios *TermiosPointer;
```

Description

The **tcgetattr** subroutine gets the parameters associated with the object referred to by the *FileDescriptor* parameter and stores them in the **termios** structure referenced by the *TermiosPointer* parameter. This subroutine is allowed from a background process; however, the terminal attributes may subsequently be changed by a foreground process.

Whether or not the terminal device supports differing input and output baud rates, the baud rates stored in the **termios** structure returned by the **tcgetattr** subroutine reflect the actual baud rates, even if they are equal.

Note: If differing baud rates are not supported, returning a value of 0 as the input baud rate is obsolete.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.
TermiosPointer	Points to a **termios** structure.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **tcgetattr** subroutine is unsuccessful if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.
ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Examples

To get the current terminal state information, enter:

```
rc = tcgetattr(stdout, &my_termios);
```

tcgetpgrp Subroutine

Purpose

Gets foreground process group ID.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

pid_t tcgetpgrp (*FileDescriptor*)
int *FileDescriptor*;

Description

The **tcgetpgrp** subroutine returns the value of the process group ID of the foreground process group associated with the terminal. The function can be called from a background process; however, the foreground process can subsequently change the information.

Parameters

Item	Description
FileDescriptor	Indicates the open file descriptor for the terminal special file.

Return Values

Upon successful completion, the process group ID of the foreground process is returned. If there is no foreground process group, a value greater than 1 that does not match the process group ID of any existing process group is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **tcgetpgrp** subroutine is unsuccessful if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* argument is not a valid file descriptor.
EINVAL	The function is not appropriate for the file associated with the *FileDescriptor* argument.
ENOTTY	The calling process does not have a controlling terminal or the file is not the controlling terminal.

tcsendbreak Subroutine

Purpose

Sends a break on an asynchronous serial data line.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <termios.h>
```

```
int tcsendbreak( FileDescriptor,  Duration)
int FileDescriptor;
int Duration;
```

Description

If the terminal is using asynchronous serial data transmission, the **tcsendbreak** subroutine causes transmission of a continuous stream of zero-valued bits for a specific duration.

If the terminal is not using asynchronous serial data transmission, the **tcsendbreak** subroutine returns without taking any action.

Pseudo-terminals and LFT do not generate a break condition. They return without taking any action.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.
Duration	Specifies the number of milliseconds that zero-valued bits are transmitted. If the value of the *Duration* parameter is 0, it causes transmission of zero-valued bits for at least 250 milliseconds and not longer than 500 milliseconds. If *Duration* is not 0, it sends zero-valued bits for *Duration* milliseconds.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **tcsendbreak** subroutine is unsuccessful if one or both of the following are true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid open file descriptor.
EIO	The process group of the writing process is orphaned, and the writing process does not ignore or block the **SIGTTOU** signal.
ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Examples

1. To send a break condition for 500 milliseconds, enter:

   ```
   rc = tcsendbreak(stdout,500);
   ```

2. To send a break condition for 25 milliseconds, enter:

   ```
   rc = tcsendbreak(1,25);
   ```

   This could also be performed using the default *Duration* by entering:

   ```
   rc = tcsendbreak(1, 0);
   ```

tcsetattr Subroutine

Purpose

Sets terminal state.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <termios.h>
```

```
int tcsetattr (FileDescriptor, OptionalActions, TermiosPointer)
int  FileDescriptor,  OptionalActions;
const struct termios *  TermiosPointer;
```

Description

The **tcsetattr** subroutine sets the parameters associated with the object referred to by the *FileDescriptor* parameter (unless support required from the underlying hardware is unavailable), from the **termios** structure referenced by the *TermiosPointer* parameter.

The value of the *OptionalActions* parameter determines how the **tcsetattr** subroutine is handled.

The 0 baud rate (B0) is used to terminate the connection. If B0 is specified as the output baud rate when the **tcsetattr** subroutine is called, the modem control lines are no longer asserted. Normally, this disconnects the line.

Using 0 as the input baud rate in the **termios** structure to cause **tcsetattr** to change the input baud rate to the same value as that specified by the value of the output baud rate, is obsolete.

If an attempt is made using the **tcsetattr** subroutine to set:

- An unsupported baud rate
- Baud rates, such that the input and output baud rates differ and the hardware does not support that combination
- Other features not supported by the hardware

but the **tcsetattr** subroutine is able to perform some of the requested actions, then the subroutine returns successfully, having set all supported attributes and leaving the above unsupported attributes unchanged.

If no part of the request can be honored, the **tcsetattr** subroutine returns a value of -1 and the **errno** global variable is set to **EINVAL**.

If the input and output baud rates differ and are a combination that is not supported, neither baud rate is changed. A subsequent call to the **tcgetattr** subroutine returns the actual state of the terminal device (reflecting both the changes made and not made in the previous **tcsetattr** call). The **tcsetattr** subroutine does not change the values in the **termios** structure whether or not it actually accepts them.

If the **tcsetattr** subroutine is called by a process which is a member of a background process group on a *FileDescriptor* associated with its controlling terminal, a **SIGTTOU** signal is sent to the background process group. If the calling process is blocking or ignoring **SIGTTOU** signals, the process performs the operation and no signal is sent.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.

Item	Description
OptionalActions	Specifies one of the following values: **TCSANOW** The change occurs immediately. **TCSADRAIN** The change occurs after all output written to the object referred to by *FileDescriptor* has been transmitted. This function should be used when changing parameters that affect output. **TCSAFLUSH** The change occurs after all output written to the object referred to by *FileDescriptor* has been transmitted. All input that has been received but not read is discarded before the change is made.
TermiosPointer	Points to a **termios** structure.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **tcsetattr** subroutine is unsuccessful if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.
EINTR	A signal interrupted the **tcsetattr** subroutine.
EINVAL	The *OptionalActions* argument is not a proper value, or an attempt was made to change an attribute represented in the **termios** structure to an unsupported value.
EIO	The process group of the writing process is orphaned, and the writing process does not ignore or block the **SIGTTOU** signal.
ENOTTY	The file associated with the *FileDescriptor* parameter is not a terminal.

Example

To set the terminal state after the current output completes, enter:

```
rc = tcsetattr(stdout, TCSADRAIN, &my_termios);
```

tcsetpgrp Subroutine

Purpose

Sets foreground process group ID.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
int tcsetpgrp ( FileDescriptor,  ProcessGroupID)
int FileDescriptor;
pid_t ProcessGroupID;
```

Description

If the process has a controlling terminal, the **tcsetpgrp** subroutine sets the foreground process group ID associated with the terminal to the value of the *ProcessGroupID* parameter. The file associated with the *FileDescriptor* parameter must be the controlling terminal of the calling process, and the controlling terminal must be currently associated with the session of the calling process. The value of the *ProcessGroupID* parameter must match a process group ID of a process in the same session as the calling process.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.
ProcessGroupID	Specifies the process group identifier.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

This function is unsuccessful if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid file descriptor.
EINVAL	The *ProcessGroupID* parameter is invalid.
ENOTTY	The calling process does not have a controlling terminal, or the file is not the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
EPERM	The *ProcessGroupID* parameter is valid, but does not match the process group ID of a process in the same session as the calling process.

termdef Subroutine

Purpose

Queries terminal characteristics.

Library

Standard C Library (**libc.a**)

Syntax

```
char *termdef ( FileDescriptor,  Characteristic)
int FileDescriptor;
char Characteristic;
```

Description

The **termdef** subroutine returns a pointer to a null-terminated, static character string that contains the value of a characteristic defined for the terminal specified by the *FileDescriptor* parameter.

Asynchronous Terminal Support

Shell profiles usually set the **TERM** environment variable each time you log in. The **stty** command allows you to change the lines and columns (by using the *lines* and *cols* options). This is preferred over changing the **LINES** and **COLUMNS** environment variables, since the **termdef** subroutine examines the environment variables last. You consider setting **LINES** and **COLUMNS** environment variables if:

- You are using an asynchronous terminal and want to override the *lines* and *cols* setting in the **terminfo** database

  ```
  OR
  ```

- Your asynchronous terminal has an unusual number of lines or columns and you are running an application that uses the **termdef** subroutine but not an application which uses the **terminfo** database (for example, **curses**).

  This is because the curses initialization subroutine, **setupterm** (“setupterm Subroutine” on page 1921), calls the **termdef** subroutine to determine the number of lines and columns on the display. If the **termdef** subroutine cannot supply this information, the **setupterm** subroutine uses the values in the **terminfo** database.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.

Item	Description
Characteristic	Specifies the characteristic that is to be queried. The following values can be specified: **c** Causes the **termdef** subroutine to query for the number of "columns" for the terminal. This is determined by performing the following actions: 1. It requests a copy of the terminal's **winsize** structure by issuing the **TIOCGWINSZ ioctl**. If **ws_col** is not 0, the **ws_col** value is used. 2. If the **TIOCGWINSZ ioctl** is unsuccessful or if **ws_col** is 0, the **termdef** subroutine attempts to use the value of the **COLUMNS** environment variable. 3. If the **COLUMNS** environment variable is not set, the **termdef** subroutine returns a pointer to a null string. **l** Causes the **termdef** subroutine to query for the number of "lines" (or rows) for the terminal. This is determined by performing the following actions: 1. It requests a copy of the terminal's **winsize** structure by issuing the **TIOCGWINSZ ioctl**. If **ws_row** is not 0, the **ws_row** value is used. 2. If the **TIOCGWINSZ ioctl** is unsuccessful or if **ws_row** is 0, the **termdef** subroutine attempts to use the value of the **LINES** environment variable. 3. If the **LINES** environment variable is not set, the **termdef** subroutine returns a pointer to a null string. **Characters other than c or l** Cause the **termdef** subroutine to query for the "terminal type" of the terminal. This is determined by performing the following actions: 1. The **termdef** subroutine attempts to use the value of the **TERM** environment variable. 2. If the **TERM** environment variable is not set, the **termdef** subroutine returns a pointer to string set to "dumb".

Examples

1. To display the terminal type of the standard input device, enter:

```
printf("%s\n", termdef(0, 't'));
```

2. To display the current lines and columns of the standard output device, enter:

```
printf("lines\tcolumns\n%s\t%s\n", termdef(2, 'l'),
    termdef(2, 'c'));
```

Note: If the **termdef** subroutine is unable to determine a value for lines or columns, it returns pointers to null strings.

test_and_set Subroutine

Purpose

Atomically tests and sets a memory location.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/atomic_op.h>

boolean_t test_and_set (word_addr, mask)
atomic_p word_addr;
int mask;
```

Description

The **test_and_set** subroutine attempts to atomically OR the value stored at *word_addr* with the value specified by *mask*. If any bit in *mask* was already set in the value stored at *word_addr*, no update is made.

Parameters

Item	Description
word_addr	Specifies the address of the memory location to be set.
mask	Specifies the mask value to be used to set the memory location specified by *word_addr*.

Return Values

The **test_and_set** subroutine returns true if the the value stored at *word_addr* was updated. Otherwise, it returns false.

tgamma, tgammaf, tgammal, tgammad32, tgammad64, and tgammad128 Subroutines

The **tgamma**, **tgammaf**, **tgammal**, **tgammad32**, **tgammad64**, and **tgammad128** subroutines compute the **gamma** function of *x*.

An application wishing to check for error situations should set **errno** to zero and call **feclearexcept**(**FE_ALL_EXCEPT**) before calling these subroutines. Upon return, if **errno** is nonzero or **fetestexcept**(**FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW | FE_UNDERFLOW**) is nonzero, an error has occurred.

Purpose

Computes the gamma.

Syntax

```
#include <math.h>

double tgamma (x)
double x;

float tgammaf (x)
float x;

long double tgammal (x)
long double x;
_Decimal32 tgammad32 (x)
_Decimal32 x;

_Decimal64 tgammad64 (x)
_Decimal64 x;
```

```
_Decimal128 tgammad128 (x)
_Decimal128 x;
```

Description

Parameters

Item	Description
x	Specifies the value to be computed.

Return Values

Upon successful completion, the **tgamma**, **tgammaf**, **tgammal**, **tgammad32**, **tgammad64**, and **tgammad128** subroutines return **Gamma**(*x*).

If *x* is a negative integer, a domain error occurs, and either a NaN (if supported), or an implementation-defined value is returned.

If the correct value would cause overflow, a range error occurs and the **tgamma**, **tgammaf**, **tgammal**, **tgammad32**, **tgammad64**, and **tgammad128** subroutines return the value of the macro **HUGE_VAL**, **HUGE_VALF**, **HUGE_VALL**, **HUGE_VAL_D32**, **HUGE_VAL_D64**, or **HUGE_VAL_D128** respectively.

If *x* is NaN, a NaN is returned.

If *x* is +Inf, *x* is returned.

If *x* is ±0, a pole error occurs, and the **tgamma**, **tgammaf**, **tgammal**, **tgammad32**, **tgammad64**, and **tgammad128** subroutines return ±**HUGE_VAL**, ±**HUGE_VALF**, ±**HUGE_VALL**, ±**HUGE_VAL_D32**, ±**HUGE_VAL_D64**, or ±**HUGE_VAL_D128** respectively.

If *x* is -Inf, a domain error occurs, and either a NaN (if supported), or an implementation-defined value is returned.

tgetent, tgetflag, tgetnum, tgetstr, or tgoto Subroutine

Purpose

Termcap database emulation.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int tgetent
(char *bp,
const char *name);

int tgetflag
(char id[2]);

int tgetnum
(char id[2]);

char *tgetstr
(char id[2],
char **area);

char *tgoto
(char *cap,
```

```
int col,
int row);
```

Description

The **tgetent** subroutine looks up the termcap entry for *name*, The emulation ignores the buffer pointer *bp*.

The **tgetflag** subroutine gets the boolean entry for *id*.

The **tgetnum** subroutine gets the numeric entry for *id*.

The **tgetstr** subroutine gets the string entry for *id*. If *area* is not a null pointer and does not point to a null pointer, the **tgetstr** subroutine copies the string entry into the buffer pointed to by **area* and advances the variable pointed to by *area* to the first byte after the copy of the string entry.

The **tgoto** subroutine instantiates the parameters *col* and *row* into the capability cap and returns a pointer to the resulting string.

All of the information available in the terminfo database need not be available through these subroutines.

Parameters

Item	Description
bp	
name	
col	
row	
***area*	
cap	`id[2]`

Return Values

Upon successful completion, subroutines that return an integer return OK. Otherwise, they return ERR.

tgetnum Subroutine

Purpose

Returns the numeric entry for the specified **termcap** identifier.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int tgetnum( ID)
char *ID;
```

Description

The **tgetnum** subroutine returns the numeric entry for the specified **termcap** identifier. This subroutine is provided for binary compatibility with applications that use the **termcap** file.

Parameters

Item	Description
ID	Specifies the 2-character string that contains a **termcap** identifier.

Return Values

The **tgetnum** subroutine returns the numeric entry for the specified **termcap** identifier.

Item	Description
-1	Returned if the ID is not found or not numeric.

tgetstr Subroutine

Purpose

Returns the string entry for the specified **termcap** identifier.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

char *tgetstr( ID,  Area)
char *ID, **Area;
```

Description

The **tgetstr** subroutine returns the string entry for the specified **termcap** identifier. This subroutine is provided for binary compatibility with applications that use the **termcap** file.

Parameters

Item	Description
Area	Contains the string entry for the specified termcap identifier. This also is returned to the calling program.
ID	Specifies the 2-character string that contains the **termcap** identifier.

Return Values

The **tgetstr** subroutine returns the string entry for the *ID* parameter, which is a 2-character string that contains a **termcap** identifier.

Item	Description
0	Returned if ID is not found or not a string capability.

tgoto Subroutine

Purpose

Duplicates the **tparm** subroutine.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
#include <term.h>
```

```
char *tgoto( Capability,  Column,  Row)
char *Capability;
int Column, Row;
```

Description

The **tgoto** subroutine calls the **tparm** (“tparm Subroutine” on page 2196) subroutine. This subroutine is provided for binary compatibility with applications that use the **termcap** file.

Parameters

Item	Description
Capability	Specifies the termcap capability to apply the parameters to.
Column	Specifies which column to apply to the capability.
Row	Specifies which row to apply to the capability.

tigetflag, tigetnum, tigetstr, or tparm Subroutine

Purpose

Retrieves capabilities from the **terminfo** database.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <term.h>

int tigetflag(char *capname,);

int tigetnum(char *capname);

char *tigetstr(char *capname);

char *tparm(char *cap,
long p1, long p2, long p3,
long p4, long p5, long p6
long p7, long p8, long p9);
```

Description

The **tigetflag**, **tigetnum**, and **tigetstr** subroutines obtain boolean, numeric, and string capabilities, respectively, from the selected record of the terminfo database. For each capability, the value to use as capname appears in the Capname column in the table in Section 6.1.3 on page 296.

The **tparm** subroutine takes as *cap* a string capability. If *cap* is parameterised (as described in Section A.1.2 on page 313), the **tparm** subroutine resolves the parameterisation. If the parameterised string refers to parameters *%p1* through *%p9*, then the **tparm** subroutine substitutes the values of *p1* through *p9*, respectively.

Return Values

Upon successful completion, the **tigetflag**, **tigetnum**, and **tigetstr** subroutines return the specified capability. The **tigetflag** subroutine returns -1 if capname is not a boolean capability. The **tigetnum** subroutine returns -2 if capname is not a numeric capability. The **tigetstr** subroutine returns (char*)-1 if capname is not a string capability.

Upon successful completion, the **tparm** subroutine returns *str* with parameterisation resolved. Otherwise, it returns a null pointer.

Parameters

Item	Description
**capname*	
**tparm*	
long p1	
long p2	
long p3	
long p4	
long p5	
long p6	
long p7	
long p8	
long p9	

Examples

For the **tigetflag** subroutine:

To determine if erase overstrike is a defined boolean capability for the current terminal, use:

```
rc = tigetflag("eo");
```

For the **tigetnum** subroutine:

To determine if number of labels is a defined numeric capability for the current terminal, use:

```
rc = tigetnum("nlab");
```

For the **tigetstr** subroutine:

To determine if "`turn on soft labels`" is a defined string capability for the current terminal, do the following:

```
char *rc;
rc = tigetstr("smln");
```

For the **tparm** subroutine:

1. To save the escape sequence used to home the cursor in the user-defined variable home_sequence, enter:

   ```
   home_sequence = tparm(cursor_home);
   ```

2. To save the escape sequence used to move the cursor to the coordinates X=40, Y=18 in the user-defined variable move_sequence, enter:

   ```
   move_sequence = tparm(cursor_address, 18, 40);
   ```

tigetnum Subroutine

Purpose

Gets the value of terminal's numeric capability.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
#include <term.h>
```

tigetnum(*CapName*)
register char ***CapName;**

Description

The **tigetnum** subroutine returns the value of terminal's numeric capability. Use this subroutine to get a capability for the current terminal. When successful, this subroutine returns the current value of the capability specified by the *CapName* parameter. Otherwise, if it is not a numeric value, this subroutine returns -2.

Note: The **tigetnum** subroutine is a low-level routine. Use this subroutine only if your application must deal directly with the terminfo database to handle certain terminal capabilities (for example, programming function keys).

Return Values

Upon successful completion, the **tigetnum** subroutine returns the value of terminal's numeric capability.

Item	Description
-2	Indicates the value specified by the *CapName* parameter is not numeric.

Parameters

Item	Description
CapName	Identifies the terminal capability to check for.

Example

To determine if number of labels is a defined numeric capability for the current terminal, use:

```
rc = tigetnum("nlab");
```

tigetstr Routine

Purpose

Returns the value of a terminal's string capability.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
#include <term.h>
```

tigetstr(*Capname***)**
register char ***Capname;**

Description

The **tigetstr** subroutine returns the value of terminal's string capability. Use this subroutine to get a capability for the current terminal pointed to by **cur_term**. When successful, this subroutine returns the current value of the capability specified by the *Capname* parameter. Otherwise, if it is not a string value, this subroutine returns (**char***) -1.

Note: The **tigetstr** subroutine is a low-level routine. Use this subroutine only if your application must deal directly with the terminfo database to handle certain terminal capabilities (for example, programming function keys).

Parameters

Item	Description
Capname	Identifies the terminal capability to check.

Example

To determine if "`turn on soft labels`" is a defined string capability for the current terminal, do the following:

```
char *rc;
```

```
rc = tigetstr("smln");
```

Return Values

Upon successful completion, the **tigetstr** subroutine returns the value of terminal's string capability.

Item	Description
(char *)-1	Indicates the value specified by the *Capname* parameter is not a string.

Files

Item	Description
/usr/include/curses.h	Contains C language subroutines and define statements for curses.

timer_create Subroutine

Purpose

Creates a per process timer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>

int timer_create (clock_id, evp, timerid)
clockid_t clock_id;
struct sigevent *evp;
timer_t *timerid;
```

Description

The **timer_create** subroutine creates a per-process timer using the specified clock, *clock_id*, as the timing base. The **timer_create** subroutine returns, in the location referenced by *timerid*, a timer ID of type **timer_t** used to identify the timer in timer requests. This timer ID is unique within the calling process until the timer is deleted. The particular clock, *clock_id*, is defined in the **time.h** file. The timer whose ID is returned is in a disarmed state upon return from the **timer_create** subroutine.

The *evp* parameter, if non-NULL, points to a **sigevent** structure. This structure, allocated by the application, defines the asynchronous notification that will occur when the timer expires. If the *evp* parameter is NULL, the effect is as if the *evp* parameter pointed to a **sigevent** structure with the **sigev_notify** member having the value **SIGEV_SIGNAL**, the **sigev_signo** member having the **SIGALARM** default signal number, and the **sigev_value** member having the value of the timer ID.

This system defines a set of clocks that can be used as timing bases for per-process timers. Supported values for the *clock_id* parameter are the following:

Item	Description
CLOCK_REALTIME	The system-wide realtime clock.
CLOCK_MONOTONIC	The system-wide monotonic clock. The value of this clock represents the amount of time since an unspecified point in the past. It cannot be set through the **clock_settime** subroutine and cannot have backward clock jumps.
CLOCK_PROCESS_CPUTIME_ID	The process CPU-time clock of the calling process. The value of this clock represents the amount of execution time of the process associated with the clock.
CLOCK_THREAD_CPUTIME_ID	The thread CPU-time clock of the calling thread. The value of this clock represents the amount of execution time of the thread associated with this clock.

The **timer_create** subroutine fails if the value defined for the *clock_id* parameter corresponds to:

- The CPU-time clock of a process that is different than the process calling the function

- The thread CPU-time clock of a thread that is different than the thread calling the function.

Parameters

Item	Description
clock_id	Specifies the clock to be used.
evp	Points to a **sigevent** structure that defines the asynchronous notification.
timerid	Points to the location where the timer ID is returned.

Return Values

If the **timer_create** subroutine succeeds, 0 is returned, and the location referenced by the *timerid* parameter is updated to a **timer_t**, which can be passed to the per-process timer calls. If an error occurs, -1 is returned and **errno** is set to indicate the error.

Error Codes

The **timer_create** subroutine will fail if:

Item	Description
EAGAIN	The system lacks sufficient signal queuing resources to honor the request.
EAGAIN	The calling process has already created all of the timers it is allowed.
EINVAL	The specified clock ID is not defined.
ENOTSUP	The implementation does not support the creation of a timer attached to the CPU-time clock that is specified by the *clock_id* parameter and associated with a process or a thread that is different from the process or thread calling **timer_create**.
ENOTSUP	The function is not supported with checkpoint-restart processes.

timer_delete Subroutine

Purpose

Deletes a per process timer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>

int timer_delete (timerid)
timer_t timerid;
```

Description

The **timer_delete** subroutine deletes the specified timer, *timerid*, that was previously created by the **timer_create** subroutine. If the timer is armed when the **timer_delete** subroutine is called, the timer is automatically disarmed before removal.

Parameters

Item	Description
timerid	Specifies the timer ID.

Return Values

If successful, the **timer_delete** subroutine returns a value of zero. Otherwise, the subroutine returns a value of -1 and sets **errno** to indicate the error.

Error Codes

The **timer_delete** subroutine fails if:

Item	Description
EINVAL	The *timerid* parameter is not a valid timer ID.
ENOTSUP	The function is not supported with checkpoint-restart processes.

timer_getoverrun, timer_gettime, and timer_settime Subroutine

Purpose

Per-process timers.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <time.h>

int timer_getoverrun (timerid)
timer_t timerid;

int timer_gettime (timerid, value)
timer_t timerid;
struct itimerspec *value;

int timer_settime (timerid, flags, value, ovalue)
timer_t timerid;
int flags;
const struct itimerspec *value;
struct itimerspec *ovalue;
```

Description

The **timer_gettime** subroutine stores the amount of time until the specified timer, *timerid*, expires, and stores the reload value of the timer into the space pointed to by the *value* parameter. The **it_value** member of the structure contains the amount of time before the timer expires, or zero if the timer is disarmed. This value is returned as the interval until the timer expires, even if the timer was armed with absolute time. The **it_interval** member of the *value* parameter contains the reload value last set by the **timer_settime** subroutine.

The **timer_settime** subroutine sets the time until the next expiration of the timer specified by the *timerid* parameter and arms the timer if the **it_value** member of the *value* parameter is nonzero. If the specified timer is armed when the **timer_settime** subroutine is called, the call resets the time until next expiration to the value specified. If the **it_value** member of the *value* parameter is zero, the timer is disarmed.

If the **TIMER_ABSTIME** flag is not set in the *flags* parameter, the **timer_settime** subroutine behaves as if the time until next expiration is set to be equal to the interval specified by the **it_value** member of the *value* parameter. That is, the timer expires in **it_value** nanoseconds from when the call is made. If the **TIMER_ABSTIME** flag is set in the *flags* parameter, the **timer_settime** subroutine behaves as if the time until next expiration is set to be equal to the difference between the absolute time specified by the **it_value** member and the current value of the clock associated with the *timerid* parameter. That is, the timer expires when the clock reaches the value specified by the **it_value** member. If the specified time has already passed, the subroutine succeeds and the expiration notification is made.

The reload value of the timer is set to the value specified by the **it_interval** member of the *value* parameter. When a timer is armed with a nonzero **it_interval**, a periodic (or repetitive) timer is specified.

Time values that are between two consecutive non-negative integer multiples of the resolution of the specified timer is rounded up to the larger multiple of the resolution. Quantization error does not cause the timer to expire earlier than the rounded time value.

If the *ovalue* parameter is not NULL, the **timer_settime** subroutine stores a value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed, together with the previous timer reload value. Timers do not expire before their scheduled time.

Only a single signal is queued to the process for a given timer at any point in time. When a timer for which a signal is still pending expires, no signal is queued, and a timer overrun occurs.

Concerning timers based on thread CPU-time clocks, the **timer_gettime** and **timer_settime** subroutines can only be called with *timerid* referencing a timer based on the thread CPU-time clock of the calling thread. In other words, a thread cannot manipulate the thread CPU-time timers created by other threads in the same process.

Parameters

Item	Description
timerid	Specifies the timer ID.
value	Points to an **itimerspec** structure containing the time value.
flags	Specifies the flags that are set.
ovalue	Specifies the location of the value representing the previous amount of time before the timer would have expired, or zero if the timer was disarmed.

Return Values

If the **timer_getoverrun** subroutine succeeds, it returns the timer expiration overrun count.

If the **timer_gettime** or t**imer_settime** subroutines succeed, 0 is returned.

If an error occurs for any of these subroutines, -1 is returned and **errno** is set to indicate the error.

Error Codes

The **timer_getoverrun**, **timer_gettime**, and **timer_settime** subroutines fail if:

Item	Description
EINVAL	The *timerid* parameter does not correspond to an ID returned by the **timer_create** subroutine but not yet deleted by the **timer_delete** subroutine.
ENOTSUP	The function is not supported with checkpoint-restart processes.

The **timer_gettime** and **timer_settime** subroutines fail if:

Item	Description
EINVAL	The *timerid* parameter corresponds to a timer based on the thread CPU-time clock of a thread different from the thread calling **timer_gettime** or **timer_settime**. The timer has not been created by this thread.

The **timer_settime** subroutine fails if:

Item	Description
EINVAL	The *value* parameter specified a nanosecond value less than zero or greater than or equal to 1000 million, and the **it_value** member of the structure did not specify zero seconds and nanoseconds.

times Subroutine

Purpose

Gets process and waited-for child process times

Syntax

```
#include <sys/times.h>

clock_t times (buffer)
struct tms *buffer;
```

Description

The **times** subroutine fills the **tms** structure pointed to by *buffer* with time-accounting information. The **tms** structure is defined in <**sys/times.h**>.

All times are measured in terms of the number of clock ticks used.

The times of a terminated child process is included in the *tms_cutime* and *tms_cstime* elements of the parent when the **wait** or **waitpid** subroutine returns the process ID of the terminated child. If a child process has not waited for its children, their times are not included in its times.

- The **tms_utime** structure member is the CPU time charged for the execution of user instructions of the calling process.
- The **tms_stime** structure member is the CPU time charged for execution by the system on behalf of the calling process.
- The **tms_cutime** structure member is the sum of the **tms_utime** and **tms_cutime** times of the child processes.
- The **tms_cstime** structure member is the sum of the **tms_stime** and **tms_cstime** times of the child processes.

Applications should use `sysconf(_SC_CLK_TCK)` to determine the number of clock ticks per second as it may vary from system to system.

Parameters

Item	Description
**buffer*	Points to the **tms** structure.

Return Values

Upon successful completion, the **times** subroutine returns the elapsed real time, in clock ticks, since an arbitrary point in the past (for example, system startup time). This point does not change from one invocation of the **times** subroutine within the process to another. The return value may overflow the possible range of type *clock_t*. If the **times** subroutine fails, `(clock_t)-1` is returned, and the **errno** global variable is set to indicate the error.

Examples

Timing a Database Lookup

The following example defines two functions, **start_clock** and **end_clock**, that are used to time a lookup. It also defines variables of type **clock_t** and **tms** to measure the duration of transactions. The **start_clock** function saves the beginning times given by the **times** subroutine. The **end_clock** function gets the ending times and prints the difference between the two times.

```
#include <sys/times.h>
#include <stdio.h>
...
void start_clock(void);
void end_clock(char *msg);
...
static clock_t st_time;
static clock_t en_time;
static struct tms st_cpu;
static struct tms en_cpu;
...
void
start_clock()
{
        st_time = times(&st_cpu);
}

/* This example assumes that the result of each subtraction is within the range of values that
can
       be represented in an integer type. */
void
end_clock(char *msg)
{
        en_time = times(&en_cpu);

        fputs(msg,stdout);
        printf("Real Time: %jd, User Time %jd, System Time %jd\n",
              (intmax_t)(en_time - st_time),
              (intmax_t)(en_cpu.tms_utime - st_cpu.tms_utime),
              (intmax_t)(en_cpu.tms_stime - st_cpu.tms_stime));
 }
```

timezone Subroutine

Attention: Do not use the **tzset** subroutine, when linking the **libc.a** and **libbsd.a** libraries. The **tzset** subroutine uses the **timezone** global external variable that conflicts with the **timezone** subroutine in the **libbsd.a** library. This name collision can cause unpredictable results.

Purpose

Returns the name of the time zone that is associated with the first parameter.

Library

Berkeley compatibility library (**libbsd.a**) (for the **timezone** subroutine only)

Syntax

```
#include <time.h>
char *timezone(zone, dst)
```

```
int zone;
int dst;
```

```
#include <time.h>
#include <limits.h>
int zone;
int dst;
```

Description

The **timezone** subroutine returns the name of the time zone that is associated with the *zone* parameter. The *zone* parameter is measured in minutes westward from Greenwich. If the **TZ** environment variable is set, the *zone* parameter is ignored, and the current time zone is calculated from the value of the **TZ** environment variable. If the value of the *dst* parameter is 0, the standard name is returned; otherwise the name of daylight saving time is returned. If the **TZ** environment variable is not set, the internal table is searched for a matching time zone. If the time zone does not appear in the built in table, the difference from GMT is produced.

The **timezone** subroutine returns a pointer to static data, which will be overwritten by subsequent calls.

Parameters

Item	Description
zone	Specifies minutes westward from Greenwich.
dst	Specifies whether to return standard time or daylight saving time.

Return Values

The **timezone** subroutine returns a pointer to the *czone* global variable, which contains the name of the time zone.

thread_cputime Subroutine

Purpose

Retrieves CPU usage for a specified thread

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/thread.h>
int thread_cputime (tid, ctime)
tid_t tid;
thread_cputime_t * ctime ;
typedef struct {
    uint64_t utime; /* User time in nanosenconds */
    uint64_t stime; /* System time in nanoseconds */
} thread_cputime_t;
```

Description

The **thread_cputime** subroutine allows a thread to query the CPU usage of the specified thread (*tid*) in the same process or in another process. If a value of -1 is passed in the *tid* parameter field, then the CPU usage of the calling thread is retrieved.

CPU usage is not the same as the total life of the thread in real time, rather it is the actual amount of CPU time consumed by the thread since it was created. The CPU usage retrieved by this subroutine contains the CPU time consumed by the requested thread *tid* in user space (*utime*) and system space (*stime*).

The thread to be queried is identified using the kernel thread ID which has global scope. This can be obtained by the application using the **thread_self** system call. Only 1:1 thread mode is supported. The result for M:N thread mode is undefined.

The CPU usage of a thread that is not the calling thread will be current as of the last time the thread was dispatched. This value will be off by a small amount if the target thread is currently running.

Parameters

Item	Description
tid	Identifier of thread for which CPU usage is to be retrieved. A value of -1 will cause the CPU usage of the calling thread to be retrieved.
ctime	CPU usage returned to the caller. The CPU usage is returned in terms of nanoseconds of system and user time.

Return Values

0
thread_cputime was successful

-1
thread_cputime was unsuccessful. Global variable errno is set to indicate the error.

Error Codes

The **thread_cputime** subroutine is unsuccessful if one or more of the following is true:

Item	Description
ESRCH	The target thread could not be found.
EINVAL	One or more of the arguments had an invalid value.
EFAULT	A copy operation to *ctime* failed.

Note: If *tid* is -1 i.e., the CPU usage of the calling thread is being requested and the *ctime* buffer is invalid, no error is returned. A SIGSEGV will be generated and the calling application will dump core.

Example

```
#include <stdio.h>
#include <sys/thread.h>
cputime.c:
int main( int argc, char *argv[])
{
    thread_cputime_t ut;
    tid_t tid;
    tid = atoi(argv[1]);
    printf("tid = %d\n",tid);
    if (thread_cputime(tid, &ut) == -1)
    {
        perror("Error from thread_cputime");
        exit(0);
    }
    else
    {
        printf("U: %ld nsecs\n", ut.utime);
        printf("S: %ld nsecs\n", ut.stime);
    }
}

Output:
```

```
# tcpdump -i en0 > /dev/null &
# echo "th * | grep tcpdump" | kdb | grep tcpdump
(0)> th * | grep tcpdump
pvthread+00A700 167 tcpdump SLEEP 0A7011 044 0 0 nethsque+000290
# echo "ibase=16;obase=A;0A7011" | bc
684049
# ./cputime 684049
tid = 684049
U: 31954040 nsecs
S: 31833069 nsecs
```

thread_post Subroutine

Purpose

Posts a thread of an event completion.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/thread.h>

int thread_post( tid)
tid_t  tid;
```

Description

The **thread_post** subroutine posts the thread whose thread ID is indicated by the value of the *tid* parameter, of the occurrence of an event. If the posted thread is waiting in **thread_wait**, it will be awakened immediately. If it not waiting in **thread_wait**, the next call to **thread_wait** does not block but returns with success immediately.

Multiple posts to the same thread without an intervening wait by the specified thread will only count as a single post. The posting remains in effect until the indicated thread calls the **thread_wait** subroutine upon which the posting gets cleared.

The **thread_wait** and the **thread_post** subroutine can be used by applications to implement a fast IPC mechanism between threads in different processes.

Parameters

Item	Description
tid	Specifies the thread ID of the thread to be posted.

Return Values

On successful completion, the **thread_post** subroutine returns a value of **0**. If unsuccessful, a value of **-1** is returned and the global variable **errno** is set to indicate the error.

Error Codes

Item	Description
ESRCH	This indicated thread is non-existent or the thread has exited or is exiting.

Item	Description
EPERM	The real or effective user ID does not match the real or effective user ID of the thread being posted, or else the calling process does not have root user authority.

thread_post_many Subroutine

Purpose

Posts one or more threads of an event completion.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/thread.h>

int thread_post_many( nthreads, tidp, erridp)
int nthreads;
tid_t * tidp;
tid_t * erridp;
```

Description

The **thread_post_many** subroutine posts one or more threads of the occurrence of the event. The number of threads to be posted is specified by the value of the *nthreads* parameter, while the *tidp* parameter points to an array of thread IDs of threads that need to be posted. The subroutine works just like the **thread_post** subroutine but can be used to post to multiple threads at the same time.

A maximum of 512 threads can be posted in one call to the **thread_post_many** subroutine.

An optional address to a thread ID field may be passed in the *erridp* parameter. This field is normally ignored by the kernel unless the subroutine fails because the calling process has no permissions to post to any one of the specified threads. In this case, the kernel posts all threads in the array pointed at by the *tidp* parameter up to the first failing thread and fills the *erridp* parameter with the failing thread's ID.

Parameters

Item	Description
nthreads	Specifies the number of threads to be posted.
tidp	Specifies the address of an array of thread IDs corresponding to the list of threads to be posted.
erridp	Either NULL or specifies the pointer to a thread ID variable in which the kernel will return the thread ID of the first failing thread when an **errno** of **EPERM** is set.

Return Values

On successful completion, the **thread_post_many** subroutine returns a value of **0**. If unsuccessful, a value of **-1** is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **thread_post_many** subroutine is unsuccessful when one of the following is true:

Item	Description
ESRCH	None of the indicated threads are existent or they have all exited or are exiting.
EPERM	The real or effective user ID does not match the real or effective user ID of one or more threads being posted, or else the calling process does not have root user authority.
EFAULT	The *tidp* parameter points to a location outside of the address space of the process.
EINVAL	A negative value or a value greater than 512 was was specified in the *nthreads* parameter.

thread_self Subroutine

Purpose

Returns the caller's kernel thread ID.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/thread.h>
```

```
tid_t thread_self ()
```

Description

The **thread_self** subroutine returns the caller's kernel thread ID. The kernel thread ID may be useful for the **bindprocessor** and **ptrace** subroutines. The **ps**, **trace**, and **vmstat** commands also report kernel thread IDs, thus this subroutine can be useful for debugging multi-threaded programs.

The kernel thread ID is unrelated with the thread ID used in the threads library (**libpthreads.a**) and returned by the **pthread_self** subroutine.

Return Values

The **thread_self** subroutine returns the caller's kernel thread ID.

thread_setsched Subroutine

Purpose

Changes the scheduling policy and priority of a kernel thread.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/sched.h>
#include <sys/pri.h>
#include <sys/types.h>

int thread_setsched ( tid,  priority,  policy)
tid_t tid;
int priority;
int policy;
```

Description

The **thread_setsched** subroutine changes the scheduling policy and priority of a kernel thread. User threads (pthreads) have their own scheduling attributes that in some cases allow a pthread to execute on top of multiple kernel threads. Therefore, if the policy or priority change is being granted on behalf of a pthread, then the pthreads contention scope should be **PTHREAD_SCOPE_SYSTEM**.

Note: Caution must be exercised when using the **thread_setsched** subroutine, since improper use may result in system hangs. See **sys/pri.h** for restrictions on thread priorities.

Parameters

Item	Description
tid	Specifies the kernel thread ID of the thread whose priority and policy are to be changed.
priority	Specifies the priority to use for this kernel thread. The priority parameter is ignored if the policy is being set to **SCHED_OTHER**. The priority parameter must have a value in the range 0 to **PRI_LOW**. **PRI_LOW** is defined in **sys/pri.h**. See **sys/pri.h** for more information on thread priorities.
policy	Specifies the policy to use for this kernel thread. The policy parameter can be one of the following values, which are defined in **sys/sched.h**: **SCHED_OTHER** Default operating system scheduling policy. **SCHED_FIFO** First in-first out scheduling policy. **SCHED_FIFO2** Allows a thread that sleeps for a relatively short amount of time to be requeued to the head, rather than the tail, of its priority run queue. **SCHED_FIFO3** Causes threads to be enqueued to the head of their run queues. **SCHED_FIFO4** This is the first in-first out scheduling policy with weak preemption. The existing running thread is not preempted by a higher priority **SCHED_FIFO4** thread unless that thread has a priority that is more than one better than the existing thread. **SCHED_RR** Round-robin scheduling policy.

Return Values

Upon successful completion, the **thread_setsched** subroutine returns a value of zero. If the **thread_setsched** subroutine is unsuccessful, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **thread_setsched** subroutine is unsuccessful if one or more of the following is true:

Item	Description
ESRCH	The kernel thread id *tid* is invalid.
EINVAL	The policy or priority is invalid.
EPERM	The caller does not have enough privilege to change the policy or priority.

thread_sigsend Subroutine

Purpose

Sends a signal to the specified thread.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/thread.h>

int thread_sigsend (tid, signal)
tid_t tid;
int signal;
```

Description

The **thread_sigsend** subroutine allows a thread in one process to send a signal to a specific thread in the same or another process. If a value of -1 is passed in the tid parameter field, the signal will be delivered to the calling thread.

The thread to receive the signal is identified by the kernel thread ID which has global scope. This can be obtained by the application using the **thread_self** system call. Only 1:1 thread mode is supported. The result for M:N thread mode is undefined.

Sending a signal number of 0 will cause only error checking to be performed. No signal be delivered to the target thread.

The effect of a signal will be same as in the case of *kill()* or *pthread_kill()* system calls, as explained in the **sigaction** section available at the *IBM Power Systems servers and AIX Information Center*.

To send a signal to a thread in another process, the real or the effective user ID of the sending process must match the real or effective user ID of the receiving process. Alternatively, if the sending process has root user authority or the **ACT_P_SIGPRIV** privilege, the sending process may send a signal to any thread. In case of insufficient privileges, an **EPERM** is returned in the global *errno* variable.

Parameters

tid
Identifier of thread that will receive the signal. A value of -1 will cause the signal to be delivered to the calling thread.

signal
The effect of a signal will be same as in the case of kill() or pthread_kill() system calls, as explained in the **sigaction** section available at the *IBM Power Systems servers and AIX Information Center*.

Return Values

0
The **thread_sigsend** was successful.

-1
The **thread_sigsend** was unsuccessful. Global variable **errno** is set to indicate the error.

Error Codes

EPERM
The thread issuing the signal does not have sufficient privileges to send the signal to the target thread.

ESRCH
The target thread could not be found.

EINVAL
Invalid signal number.

Example

mykill.c :

```
#include <sys/thread.h>
#include <sys/signal.h>
int main(int argc, char *argv[])
{
    int rc, sig;
    tid_t tid;
    if (argc < 3) {
       printf("Syntax: %s <tid> <signo>\n", argv[0]);
       exit(0);
}
    tid = atoi(argv[1]);
    sig = atoi(argv[2]);
    if (thread_sigsend(tid, signo) == -1)
       perror("thread_sigsend returned error");
       printf("Sent signal %d to thread %d\n",sig,tid);
}
```

mythread.c :

```
#include <stdio.h>
#include <signal.h>
#include <pthread.h>
void *thread_func(void *);
void sighand(int signo)
{
      printf("-- Received signal %d in thread %d\n",
      signo, thread_self());
}
int main(int argc, char *argv[])
{
    int rc,i,signo;
    pthread_t *ptid;
    struct sigaction actions;
    int numthreads;
    if (argc < 3) {
       printf("Syntax: %s <numthreads> <signo>\n", argv[0]);
       exit(0);
    }
    numthreads = atoi(argv[1]);
        if (numthreads < 1)
        numthreads = 1;
    signo = atoi(argv[2]);
    ptid = (pthread_t *)calloc(1,
    numthreads*sizeof(pthread_t));
    pthread_init();
    memset(&actions, 0, sizeof(actions));
    sigemptyset(&actions.sa_mask);
    actions.sa_flags = 0;
    actions.sa_handler = sighand;
    rc = sigaction(signo,&actions,NULL);
    for (i=0; i<numthreads; i++) {
        rc = pthread_create(&ptid[i],NULL,thread_func, NULL);
           if (rc != 0) {
               printf("pthread_create func1 failed. rc =
                  %d\n",rc);
```

```
                    exit(-1);
                }
            }
    for (i=0; i<numthreads; i++)
       pthread_join(ptid[i],NULL);
    free(ptid);
}
void *thread_func(void *p)
{
    int rc;
    tid_t tid = thread_self();
    printf("Thread %d started\n", tid);
    rc = sleep(20);
    if (rc != 0) {
       printf("tid %d woken up with rc %d, errno %d\n",
          tid, rc, errno);
       eturn NULL;
    }
       printf("tid %d completed sleep\n", tid);
       pthread_exit(NULL);
}
Output:
# ./mythread 3 30 &
[1] 192734
Thread 684281 started
Thread 786593 started
Thread 1101959 started
# ./mykill 786593
Sent signal 30 to 786593
-- Received signal 30 in thread 786593
tid 786593 woken up with rc 15, errno 0
# ./mykill 684281
Sent signal 30 to 684281
-- Received signal 30 in thread 684281
tid 684281 woken up with rc 9, errno 0
# tid 1101959 completed sleep
```

thread_wait Subroutine

Purpose

Suspends the thread until it receives a post or times out.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/thread.h>

int thread_wait( timeout)
int  timeout;
```

Description

The **thread_wait** subroutine allows a thread to wait or block until another thread posts it with the **thread_post** or the **thread_post_many** subroutine or until the time limit specified by the *timeout* value expires. It returns immediately if there is a pending post for this thread or if a *timeout* value of 0 is specified.

If the event for which the thread is waiting and for which it will be posted will occur only in the future, the **thread_wait** subroutine may be called with a *timeout* value of **0** to clear any pending posts.

The **thread_wait** and the **thread_post** subroutine can be used by applications to implement a fast IPC mechanism between threads in different processes.

Parameters

Item	Description
timeout	Specifies the maximum length of time, in milliseconds, to wait for a posting. If the *timeout* parameter value is **-1**, the **thread_wait** subroutine does not return until a posting actually occurs. If the value of the *timeout* parameter is **0**, the **thread_wait** subroutine does not wait for a post to occur but returns immediately, even if there are no pending posts. For a non-privileged user, the minimum *timeout* value is 10 msec and any value less than that is automatically increased to 10 msec.

Return Values

On successful completion, the **thread_wait** subroutine returns a value of **0**. The **thread_wait** subroutine completes successfully if there was a pending post or if the calling thread was posted before the time limit specified by the *timeout* parameter expires.

A return value of **THREAD_WAIT_TIMEDOUT** indicates that the **thread_wait** subroutine timed out.

If unsuccessful, a value of **-1** is returned and the global variable **errno** is set to indicate the error.

Error Codes

The **thread_wait** subroutine is unsuccessful when one of the following is true:

Item	Description
EINTR	This subroutine was terminated by receipt of a signal.
ENOMEM	There is not enough memory to allocate a timer

thrd_create Subroutine

Purpose

This subroutine creates a thread.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
int thrd_create(thrd_t *thr, thrd_start_t func, void *arg);
```

Description

The **thrd_create** subroutine creates a new thread by running the **func(arg)** subroutine. If the **thrd_create** subroutine succeeds, it sets the object specified by the `thr` parameter to the identifier of the newly created thread.

Notes:

- A thread's identifier can be reused for a different thread after the original thread is exited and the thread is detached or joined to another thread.
- The completion of the **thrd_create** subroutine synchronizes with the starting of the new thread.

Parameters

status

Item	Description
thr	Holds the identifier of the newly created thread.
func	Specifies the subroutine that is used to create a new thread.
arg	Specifies the argument to the **func()** subroutine.

Return Values

The **thrd_create** subroutine returns **thrd_success** on success, **thrd_nomem** if no memory is allocated for the thread that is requested, or **thrd_error** if the request is not completed.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

thrd_current Subroutine

Purpose

This subroutine identifies the thread that is being requested.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
thrd_t thrd_current(void);
```

Description

The **thrd_current** subroutine identifies the thread that is being requested.

Parameters

None

Return Values

The **thrd_current** subroutine returns the identifier of the thread that is being requested.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

thrd_detach Subroutine

Purpose

This subroutine detaches the `thr` thread.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
int thrd_detach(thrd_t thr);
```

Description

The **thrd_detach** subroutine instructs the operating system to return any resources that are allocated to the thread identified by the `thr` parameter during the thread termination. The thread that is identified by the `thr` parameter is not a previously detached thread or a joined thread with another thread.

Parameters

Item	Description
`thr`	Holds the identifier of the newly created thread.

Return Values

The **thrd_detach** subroutine returns **thrd_success** on successful completion or it returns **thrd_error** if the request does not complete.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

thrd_equal Subroutine

Purpose

This subroutine compares two threads.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
int thrd_equal(thrd_t thr0, thrd_t thr1);
```

Description

The **thrd_equal** subroutine determines whether the thread identified by the `thr0` parameter refers to the thread identified by the `thr1` parameter.

Parameters

Item	Description
`thr0`	Refers to the first thread to be compared.
`thr1`	Refers to the second thread to be compared.

Return Values

The **thrd_equal** subroutine returns zero if the `thr0` thread and the `thr1` thread refer to different threads. Otherwise, the **thrd_equal** subroutine returns a nonzero value.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

thrd_exit Subroutine

Purpose

This subroutine ends the thread from running.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
_Noreturn void thrd_exit(int res);
```

Description

The **thrd_exit** subroutine ends the calling thread from running and sets its result code to **res**.

The program ends normally after the last thread is stopped. The behavior is the same as if the program called the **exit** subroutine with the **EXIT_SUCCESS** status when the thread ends.

Parameters

Item	Description
res	Holds the result code of the calling thread.

Return Values

The **thrd_exit** subroutine returns no value.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

thrd_join Subroutine

Purpose

This subroutine joins the thread that is identified by the **thr** parameter and updates the **res** parameter with the results.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
int thrd_join(thrd_t thr, int *res);
```

Description

The **thrd_join** subroutine joins the thread that is identified by the **thr** parameter with the current thread by blocking until the other thread is stopped. If the **res** parameter is not a null pointer, it stores the thread's result code in the integer specified by the **res** parameter. The ending of the other thread is synchronized with the completion of the **thrd_join** subroutine. The thread that is identified by the **thr** parameter is not previously detached or joined with another thread.

Parameters

Item	Description
thr	Specifies the thread that must be joined with the current thread.
res	Holds the thread's result code if the value specified is not a null pointer.

Return Values

The **thrd_join** subroutine returns **thrd_success**on successful completion or it returns **thrd_error** if the request is not completed.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

thrd_sleep Subroutine

Purpose

This subroutine causes the thread to sleep or pause until a time interval duration elapses.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
int thrd_sleep(const struct timespec *duration, struct timespec *remaining);
```

Description

The **thrd_sleep** subroutine suspends running of the calling thread until either the interval specified by the **duration** parameter elapses or a signal which is not being ignored, is received. If interrupted by a signal and the **remaining** argument is not null, the amount of remaining time (the requested interval minus the time actually slept) is stored in the interval it points to. The **duration** and **remaining** arguments potentially point to the same object. The suspension time is longer than requested because the interval is rounded up to an integer multiple of the sleep resolution or because of the scheduling of other activities by the system. However, when the thread is interrupted by a signal, the suspension time is not less than the specified time, as measured by the **TIME_UTC** system clock .

Parameters

Item	Description
duration	Specifies the number of time intervals for which (or until a signal is received) a calling thread is suspended.
remaining	Specifies the amount of remaining time (the requested interval minus the time actually slept).

Return Values

The **thrd_sleep** subroutine returns zero if the requested time elapses. The **thrd_sleep** subroutine returns -1 if it is interrupted by a signal. The **thrd_sleep** subroutine returns a negative value if it fails to complete.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

thrd_yield Subroutine

Purpose

This subroutine yields to other threads and allows them to run first.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
void thrd_yield(void);
```

Description

The **thrd_yield** subroutine allows other threads to run, even if the current thread continues to run.

Parameters

None

Return Values

The **thrd_yield** subroutine returns no value.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

tmpfile Subroutine

Purpose

Creates a temporary file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
FILE *tmpfile ( )
```

Description

The **tmpfile** subroutine creates a temporary file and opens a corresponding stream. The file is opened for update. The temporary file is automatically deleted when all references (links) to the file have been closed.

The stream refers to a file which has been unlinked. If the process ends in the period between file creation and unlinking, a permanent file may remain.

Return Values

The **tmpfile** subroutine returns a pointer to the stream of the file that is created if the call is successful. Otherwise, it returns a null pointer and sets the **errno** global variable to indicate the error.

Error Codes

The **tmpfile** subroutine fails if one of the following occurs:

Item	Description
EINTR	A signal was caught during the **tmpfile** subroutine.
EMFILE	The number of file descriptors currently open in the calling process is already equal to **OPEN_MAX**.
ENFILE	The maximum allowable number of files is currently open in the system.
ENOSPEC	The directory or file system which would contain the new file cannot be expanded.

tmpnam or tempnam Subroutine

Purpose

Constructs the name for a temporary file.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
char *tmpnam ( String)
char *String;

char *tempnam ( Directory,  FileXPointer)
const char *Directory, *FileXPointer;
```

Description

Attention: The **tmpnam** and **tempnam** subroutines generate a different file name each time they are called. If called more than 16,384 (**TMP_MAX**) times by a single process, these subroutines recycle previously used names.

The **tmpnam** and the **tempnam** subroutines generate file names for temporary files. The **tmpnam** subroutine generates a file name using the path name defined as **P_tmpdir** in the **stdio.h** file.

Files created using the **tmpnam** subroutine reside in a directory intended for temporary use. The file names are unique. The application must create and remove the file.

The **tempnam** subroutine enables you to define the directory. The *Directory* parameter points to the name of the directory in which the file is to be created. If the *Directory* parameter is a null pointer or points to

a string that is not a name for a directory, the path prefix defined as **P_tmpdir** in the **stdio.h** file is used. For an application that has temporary files with initial letter sequences, use the *FileXPointer* parameter to define the sequence. The *FileXPointer* parameter (a null pointer or a string of up to 5 bytes) is used as the beginning of the file name.

Between the time a file name is created and the file is opened, another process can create a file with the same name. Name duplication is unlikely if the other process uses these subroutines or the **mktemp** subroutine, and if the file names are chosen to avoid duplication by other means.

Parameters

Item	Description
String	Specifies the address of an array of at least the number of bytes specified by **L_tmpnam**, a constant defined in the **stdio.h** file. If the *String* parameter has a null value, the **tmpnam** subroutine places its result into an internal static area and returns a pointer to that area. The next call to this subroutine destroys the contents of the area. If the *String* parameter's value is not null, the **tmpnam** subroutine places its results into the specified array and returns the value of the *String* parameter.
Directory	Points to the path name of the directory in which the file is to be created. The **tempnam** subroutine controls the choice of a directory. If the *Directory* parameter is a null pointer or points to a string that is not a path name for an appropriate directory, the path name defined as **P_tmpdir** in the **stdio.h** file is used. If that path name is not accessible, the **/tmp** directory is used. You can bypass the selection of a path name by providing an environment variable, **TMPDIR**, in the user's environment. The value of the **TMPDIR** environment variable is a path name for the desired temporary-file directory.
FileXPointer	A pointer to an initial character sequence with which the file name begins. The *FileXPointer* parameter value can be a null pointer, or it can point to a string of characters to be used as the first characters of the temporary-file name. The number of characters allowed is file system dependent, but 5 bytes is the maximum allowed.

Return Values

Upon completion, the **tempnam** subroutine allocates space for the string using the **malloc** subroutine, puts the generated path name in that space, and returns a pointer to the space. Otherwise, it returns a null pointer and sets the **errno** global variable to indicate the error. The pointer returned by **tempnam** may be used in the **free** subroutine when the space is no longer needed.

Error Codes

The **tempnam** subroutine returns the following error code if unsuccessful:

Item	Description
ENOMEM	Insufficient storage space is available.

Item	Description
ENINVAL	Indicates an invalid *string* value.

touchoverlap Subroutine

Purpose

Marks the overlap of two windows as changed and makes arrangements for their refresh.

Library

Curses Library (**libcurses.a**)

Syntax

#include <curses.h>

touchoverlap(*Window1*, *Window2***)**
WINDOW ****Window1*, *Window2*;**

Description

The **touchoverlap** subroutine marks the overlap of two windows as changed and makes arrangements for their refresh.

Parameters

Item	Description
Window1	Specifies the first window as changed.
Window2	Specifies the second window as changed.

Examples

To mark the overlap of the two user-defined windows `my_window` and `my_new_window` as changed, enter:

```
touchoverlap(my_window, my_new_window);
```

touchwin Subroutine

Purpose

Forces every character in a window's buffer to be refreshed at the next call to the **wrefresh** subroutine.

Library

Curses Library (**libcurses.a**)

Syntax

#include <curses.h>

touchwin(*Window***)**
WINDOW ****Window*;**

Description

The **touchwin** (“touchwin Subroutine” on page 2180) subroutine forces every character in the specified window to be refreshed during the next call to the **refresh** or **wrefresh** subroutine. To force a specific range of lines to be refreshed, use the **touchline** (**is_linetouched**, **is_wintouched**, **touchline**, **touchwin**, **untouchwin**, or **wtouchin**) subroutine.

The combined usage of the **touchwin** and **wrefresh** subroutines is helpful when dealing with subwindows or overlapping windows. When dealing with overlapping windows, it may become necessary to bring the back window to the front. A call to the **wrefresh** subroutine does not change the terminal because none of the characters in the window were changed. Calling the **touchwin** subroutine on the back window before the **wrefresh** subroutine redisplays the window on the terminal and, effectively, brings it to the front.

Parameters

Item	Description
Window	Specifies the window to be touched.

Example

To refresh a user-defined parent window, `parent_window`, that has been edited through its subwindows, use:

```
WINDOW *parent_window;
touchwin(parent_window);

wrefresh(parent_window);
```

This forces **curses** to disregard any optimization information it may have for `my_window`. **curses** assumes all lines and columns have changed for `my_window`.

towctrans, or towctrans_l Subroutine

Purpose

Character transliteration.

Library

Standard library (**libc.a**)

Syntax

```
#include <wctype.h>
```

```
wint_t towctrans (wint_t wc, wctrans_t desc);
wint_t towctrans_l (wint_t wc, wctrans_t desc, locale_t Locale);
```

Description

The **towctrans** and **towctrans_l** functions transliterates the wide-character code wc using the mapping described by desc. The current setting of the LC_CTYPE category in the current locale of the process or in the locale represented by *Locale*, respectively, should be the same as during the call to **wctrans** or **wctrans_l** that returned the value *desc*. If the value of *desc* is invalid (that is, not obtained by a call to **wctrans** or *desc* is invalidated by a subsequent call to **setlocale** that has affected category LC_CTYPE or not obtained by a call to *wctrans_l* with the same locale object *Locale*) the result is implementation-dependent.

Return Values

If successful, the **towctrans**, and **towctrans_l** functions return the mapped value of wc using the mapping described by *desc*. Otherwise it returns *wc* unchanged.

Error Codes

The **towctrans**, and **towctrans_l** function may fail if:

Item	Description
EINVAL	*desc* contains an invalid transliteration descriptor.

towlower, or towlower_l Subroutine

Purpose

Converts an uppercase wide character to a lowercase wide character.

Library

Standard C Library (**libc.a**)

Syntax

#include <wchar.h>

wint_t towlower (*WC***) wint_t** *WC***;**

wint_t towlower_l(*WC*,*Locale***) wint_t** *WC***; locale_t** *Locale***;**

Description

The **towlower** subroutine converts the uppercase wide character specified by the *WC* parameter into the corresponding lowercase wide character. The **LC_CTYPE** category affects the behavior of the **towlower** subroutine.

The **towlower_l** subroutine is same as the **towlower** routine, except that the locale data used is from the locale represented by *Locale*.

Parameters

Item	Description
WC	Specifies the wide character to convert to lowercase.
Locale	Specifies the locale in which character has to be converted.

Return Values

If the *WC* parameter contains an uppercase wide character that has a corresponding lowercase wide character, that wide character is returned. Otherwise, the *WC* parameter is returned unchanged.

towupper, or towupper_l Subroutine

Purpose

Converts a lowercase wide character to an uppercase wide character.

Library

Standard C Library (**libc.a**)

Syntax

#include <wchar.h>

wint_t towupper (*WC***) wint_t** *WC***;**

wint_t towupper_l (*WC***,** *Locale***) wint_t** *WC***; locale_t** *Locale***;**

Description

The **towupper** subroutine converts the lowercase wide character specified by the *WC* parameter into the corresponding uppercase wide character. The **LC_CTYPE** category affects the behavior of the **towupper** subroutine.

The **towupper_l** subroutine is same as the **towupper** subroutine, except that the locale data used is from the locale represented by *Locale*.

Parameters

Item	Description
WC	Specifies the wide character to convert to uppercase.
Locale	Specifies the locale in which character has to be converted.

Return Values

If the *WC* parameter contains a lowercase wide character that has a corresponding uppercase wide character, that wide character is returned. Otherwise, the *WC* parameter is returned unchanged.

t_rcvreldata Subroutine

Purpose

Receive an orderly release indication or confirmation containing user data.

Library

Syntax

```
#include <xti.h>

int t_rcvreldata(
    int fd,
    struct t_discon *discon)
```

Description

This function is used to receive an orderly release indication for the incoming direction of data transfer and to retrieve any user data sent with the release. The argument *fd* identifies the local transport endpoint where the connection exists, and **discon** points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

After receipt of this indication, the user may not attempt to receive more data via **t_rcvv** ("t_rcvv Subroutine" on page 2185). Such an attempt will fail with **t_error** set to [TOUTSTATE]. However, the user may continue to send data over the connection if **t_sndreldata** ("t_sndreldata Subroutine " on page 2191) has not been called by the user.

The field *reason* specifies the reason for the disconnection through a protocol-dependent reason code, and **udata** identifies any user data that was sent with the disconnection; the field *sequence* is not used.

If a user does not care if there is incoming data and does not need to know the value of *reason*, **discon** may be a null pointer, and any user data associated with the disconnection will be discarded.

If **discon->udata.maxlen** is greater than zero and less than the length of the value, **t_rcvreldata** fails with **t_errno** set to [TBUFOVFLW].

This function may not be available on all systems.

Parameters	Before call	After call
fd	x	/
discon->	udata.maxlen	x
discon->	udata.len	/
discon->	udata.buf	?
discon->	reason	/
discon->	sequence	/

Valid States

T_DATAXFER, T_OUTREL

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **t_errno** is set to indicate an error.

Error Codes

On failure, the **t_errno** subroutine is set to one of the following:

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW
The number of bytes allocated for incoming data (**maxlen**) is greater than 0 but not sufficient to store the data, and the disconnection information to be returned in **discon** will be discarded. The provider state, as seen by the user, will be changed as if the data was successfully retrieved.

TLOOK
An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNOREL
No orderly release indication currently exists on the specified transport endpoint.

TNOTSUPPORT
Orderly release is not supported by the underlying transport provider.

TOUTSTATE
The communications endpoint referenced by **fd** is not in one of the states in which a call to this function is valid.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (**t_errno**).

TSYSERR
A system error has occurred during execution of this function.

t_rcvv Subroutine

Purpose

Receive data or expedited data sent over a connection and put the data into one or more non-contiguous buffers.

Library

libxti.*

Syntax

```
#include <xti.h>
```

```
int t_rcvv (int fd, struct t_iovec *iov, unsigned int iovcount, int *flags) ;
```

Description

This function receives either normal or expedited data. The argument *fd* identifies the local transport endpoint through which data will arrive, *iov* points to an array of buffer address/buffer size pairs (*iov_base, iov_len*). The **t_rcvv** function receives data into the buffers specified by *iov[0].iov_base*, *iov[1].iov_base*, through *iov[iovcount-1].iov_base*, always filling one buffer before proceding to the next.

Note: The limit on the total number of bytes available in all buffers passed (that is, iov(0).iov_len + . . + iov(iovcount-1).iov_len) may be constrained by implementation limits. If no other constraint applies, it will be limited by [INT_MAX]. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

The argument *iovcount* contains the number of buffers which is limited to T_IOV_MAX (an implementation-defined value of at least 16). If the limit is exceeded, the function will fail with [TBADDATA].

The argument flags may be set on return from **t_rcvv** and specifies optional flags as described below.

By default, **t_rcvv** operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NONBLOCK is set (via **t_open** or **fcntl**, **t_rcvv** will execute in asynchronous mode and will fail if no data is available (see [TNODATA] below).

On return from the call, if T_MORE is set in flags, this indicates that there is more data, and the current transport service data unit (TSDU) or expedited transport service data unit (ETSDU) must be received in multiple **t_rcvv** or **t_rcv** calls. In the asynchronous mode, or under unusual conditions (for example, the arrival of a signal or T_EXDATA event), the T_MORE flag may be set on return from the **t_rcvv** call even when the number of bytes received is less than the total size of all the receive buffers. Each **t_rcvv** with the T_MORE flag set indicates that another **t_rcvv** must follow to get more data for the current TSDU. The end of the TSDU is identified by the return of a **t_rcvv** call with the T_MORE flag not set. If the transport provider does not support the concept of a TSDU as indicated in the info argument on return from **t_open ort_getinfo** , the T_MORE flag is not meaningful and should be ignored. If the amount of buffer space passed in iov is greater than zero on the call to **t_rcvv**, then **t_rcvv** will return 0 only if the end of a TSDU is being returned to the user.

On return, the data is expedited if T_EXPEDITED is set in flags. If T_MORE is also set, it indicates that the number of expedited bytes exceeded nbytes, a signal has interrupted the call, or that an entire ETSDU was not available (only for transport protocols that support fragmentation of ETSDUs). The rest of the ETSDU will be returned by subsequent calls to **t_rcvv** which will return with T_EXPEDITED set in flags. The end of the ETSDU is identified by the return of a **t_rcvv** call with T_EXPEDITED set and T_MORE

cleared. If the entire ETSDU is not available it is possible for normal data fragments to be returned between the initial and final fragments of an ETSDU.

If a signal arrives, **t_rcvv** returns, giving the user any data currently available. If no data is available, **t_rcvv** returns -1, sets **t_errno** to [TSYSERR] and errno to [EINTR]. If some data is available, **t_rcvv** returns the number of bytes received and T_MORE is set in flags.

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for the T_DATA or T_EXDATA events using the **t_look** function. Additionally, the process can arrange to be notified via the EM interface.

Parameters	Before call	After call
fd	X	/
iov	X/	
iovcount	X	/
iov[0].iov_base	X(/)	=(X)
iov[0].iov_len	X	=
. . . .		
iov[iovcount-1].iov_base	X(/)	=(X)
iov[iovcount-1].iov_len	X	=

Return Values

On successful completion, **t_rcvv** returns the number of bytes received. Otherwise, it returns -1 on failure and **t_errno** is set to indicate the error.

Error Codes

On failure, **t_errno** is set to one of the following:

Item	Description
TBADDATA	iovcount is greater than T_IOV_MAX.
TBADF	The specified file descriptor does not refer to a transport endpoint.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNODATA	O_NONBLOCK was set, but no data is currently available from the transport provider.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by fd is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (t_errno).
TSYSERR	A system error has occurred during execution of this function.

t_rcvvudata Subroutine

Purpose

Receive a data unit into one or more noncontiguous buffers.

Library

Standard library (**libxti.a**)

Syntax

```
#include <xti.h>
 int t_rcvvudata (
     int fd, struct t_unitdata *unitdata, struct t_iovec *iov,unsigned int iovcount, int *flags)
```

Description

This function is used in connectionless mode to receive a data unit from another transport user. The argument **fd** identifies the local transport endpoint through which data will be received, **unitdata** holds information associated with the received data unit, **iovcount** contains the number of non-contiguous udata buffers which is limited to T_IOV_MAX (an implementation-defined value of at least 16), and **flags** is set on return to indicate that the complete data unit was not received. If the limit on **iovcount** is exceeded, the function fails with [TBADDATA]. The argument **unitdata** points to a **t_unitdata** structure containing the following members:

```
   struct netbuf addr;
   struct netbuf opt;
   struct netbuf udata;
```

The **maxlen** field of **addr** and **opt** must be set before calling this function to indicate the maximum size of the buffer for each. The **udata** field of **t_unitdata** is not used. The **iov_len** and **iov_base** fields of **iov[0]** through **iov[iovcount-1]** must be set before calling **t_rcvvudata** to define the buffer where the userdata will be placed. If the maxlen field of **addr** or **opt** is set to zero then no information is returned in the **buf** field for this parameter.

On return from this call, **addr** specifies the protocol address of the sending user, **opt** identifies options that were associated with this data unit, and **iov[0].iov_base** through **iov[iovcount-1]. iov_base** contains the user data that was received. The return value of **t_rcvvudata** is the number of bytes of user data given to the user.

Note: The limit on the total number of bytes available in all buffers passed (that is, **iov(0).iov_len + . . + iov(iovcount-1).iov_len**) may be constrained by implementation limits. If no other constraint applies, it will be limited by [INT_MAX]. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

By default, **t_rcvvudata** operates in synchronous mode and waits for a data unit to arrive if none is currently available. However, if O_NONBLOCK is set (via **t_open** or **fcntl**), **t_rcvvudata** executes in asynchronous mode and fails if no data units are available. If the buffers defined in the iov[] array are not large enough to hold the current data unit, the buffers will be filled and T_MORE will be set in flags on return to indicate that another **t_rcvvudata** should be called to retrieve the rest of the data unit. Subsequent calls to **t_rcvvudata** will return zero for the length of the address and options, until the full data unit has been received.

Parameters	Before call	After call
fd	X	/
unitdata->addr.maxlen	X	=
unitdata->addr.len	/	X
unitdata->addr.buf	?(/)	=(/)
unitdata->opt.maxlen	X	=
unitdata->opt.len	/	X
unitdata->opt.buf	?(/)	=(?)

Parameters	Before call	After call
unitdata->udata.maxlen	/	=
unitdata->udata.len	/	=
unitdata->udata.buf	/	=
iov[0].iov_base	X	=(X)
iov[0].iov_len	X	=
. . . .		
iov[iovcount-1].iov_base	X(/)	=(X)
iov[iovcount-1].iov_len	X	=
iovcoun	X	/
flags	/	/

Return Values

On successful completion, **t_rcvvudata** returns the number of bytes received. Otherwise, it returns -1 on failure and **t_errno** is set to indicate the error.

Error Codes

On failure, **t_errno** is set to one of the following:

Item	Description
TBADDATA	**iovcount** is greater than T_IOV_MAX.
TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The number of bytes allocated for the incoming protocol address or options (**maxlen**) is greater than 0 but not sufficient to store the information. The unit data information to be returned in **unitdata** will be discarded.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNODATA	O_NONBLOCK was set, but no data units are currently available from the transport provider.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by **fd** is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (**t_errno**).
TSYSERR	A system error has occurred during execution of this function.

t_sndv Subroutine

Purpose

Send data or expedited data, from one or more non-contiguous buffers, on a connection.

Library

Standard library (**libxti.a**)

Syntax

```
#include <xti.h>
int t_sndv (int fd, const struct t_iovec *iov, unsigned it iovcount, int flags)
```

Description

Parameters	Before call	After call
fd	X	/
iovec	X	/
iovcount	X	/
iov[0].iov_base	X(X)	/
iov[0].iov_len	X	/
. . . .		
iov[iovcount-1].iov_base	X(X)	/
iov[iovcount-1].iov_len	X	=
flags	X	/

This function is used to send either normal or expedited data. The argument **fd** identifies the local transport endpoint over which data should be sent, **iov** points to an array of buffer address/buffer length pairs. **t_sndv** sends data contained in buffers **iov[0]**, **iov[1]**, through **iov[iovcount-1]**. **iovcount** contains the number of non-contiguous data buffers which is limited to T_IOV_MAX (an implementation-defined value of at least 16). If the limit is exceeded, the function fails with [TBADDATA].

Note: The limit on the total number of bytes available in all buffers passed (that is: **iov(0).iov_len + . . + iov(iovcount-1).iov_len**) may be constrained by implementation limits. If no other constraint applies, it will be limited by [INT_MAX]. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

The argument **flags** specifies any optional flags described below:

T_EXPEDITED
If set in **flags**, the data will be sent as expedited data and will be subject to the interpretations of the transport provider.

T_MORE
If set in **flags**, this indicates to the transport provider that the transport service data unit (TSDU) (or expedited transport service data unit ETSDU) is being sent through multiple **t_sndv** calls. Each **t_sndv** with the T_MORE flag set indicates that another **t_sndv** (or **t_snd**) will follow with more data for the current TSDU (or ETSDU).

The end of the TSDU (or ETSDU) is identified by a **t_sndv** call with the T_MORE flag not set. Use of T_MORE enables a user to break up large logical data units without losing the boundaries of those units at the other end of the connection. The flag implies nothing about how the data is packaged for transfer below the transport interface. If the transport provider does not support the concept of a TSDU as indicated in the **info** argument on return from **t_open ort_getinfo**, the T_MORE flag is not meaningful and will be ignored if set.

The sending of a zero-length fragment of a TSDU or ETSDU is only permitted where this is used to indicate the end of a TSDU or ETSDU, that is, when the T_MORE flag is not set. Some transport providers also forbid zero-length TSDUs and ETSDUs. See "Base Operating System error codes for services that require path-name resolution" for a fuller explanation.

If set in flags, requests that the provider transmit all data that it has accumulated but not sent. The request is a local action on the provider and does not affect any similarly named protocol flag (for example, the TCP PUSH flag). This effect of setting this flag is protocol-dependent, and it may be ignored entirely by transport providers which do not support the use of this feature.

Note: The communications provider is free to collect data in a send buffer until it accumulates a sufficient amount for transmission.

By default, **t_sndv** operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NONBLOCK is set (via **t_open** or **fcntl**), **t_sndv** executes in asynchronous mode, and will fail immediately if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared via either **t_look** or the EM interface.

On successful completion, **t_sndv** returns the number of bytes accepted by the transport provider. Normally this will equal the total number of bytes to be sent, that is,

```
(iov[0].iov_len + . . + iov[iovcount-1].iov_len)
```

However, the interface is constrained to send at most INT_MAX bytes in a single send. When **t_sndv** has submitted INT_MAX (or lower constrained value, see the note above) bytes to the provider for a single call, this value is returned to the user. However, if O_NONBLOCK is set or the function is interrupted by a signal, it is possible that only part of the data has actually been accepted by the communications provider. In this case, **t_sndv** returns a value that is less than the value of nbytes. If **t_sndv** is interrupted by a signal before it could transfer data to the communications provider, it returns -1 with t_errno set to [TSYSERR] and errno set to [EINTR].

If the number of bytes of data in the iov array is zero and sending of zero octets is not supported by the underlying transport service, **t_sndv** returns -1 with t_errno set to [TBADDATA].

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as specified by the current values in the TSDU or ETSDU fields in the info argument returned by **t_getinfo**.

The error [TLOOK] is returned for asynchronous events. It is required only for an incoming disconnect event but may be returned for other events.

Return Values

On successful completion, **t_sndv** returns the number of bytes accepted by the transport provider. Otherwise, -1 is returned on failure and **t_errno** is set to indicate the error.

Note:

1. In synchronous mode, if more than INT_MAX bytes of data are passed in the iov array, only the first INT_MAX bytes will be passed to the provider.
2. If the number of bytes accepted by the communications provider is less than the number of bytes requested, this may either indicate that O_NONBLOCK is set and the communications provider is blocked due to flow control, or that O_NONBLOCK is clear and the function was interrupted by a signal.

Error Codes

On failure, t_errno is set to one of the following:

Item	Description
TBADDATA	Illegal amount of data:

- A single send was attempted specifying a TSDU (ETSDU) or fragment TSDU (ETSDU) greater than that specified by the current values of the TSDU or ETSDU fields in the **info** argument.
- A send of a zero byte TSDU (ETSDU) or zero byte fragment of a TSDU (ETSDU) is not supported by the provider.

- Multiple sends were attempted resulting in a TSDU (ETSDU) larger than that specified by the current value of the TSDU or ETSDU fields in the **info** argument the ability of an XTI implementation to detect such an error case is implementation-dependent (see CAVEATS, below).
- **iovcount** is greater than T_IOV_MAX.

Item	Description
TBADF	The specified file descriptor does not refer to a transport endpoint.
TBADFLAG	An invalid flag was specified.
TFLOW	O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.
TLOOK	An asynchronous event has occurred on this transport endpoint.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The communications endpoint referenced by **fd** is not in one of the states in which a call to this function is valid.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (**t_errno**).
TSYSERR	A system error has occurred during execution of this function.

t_sndreldata Subroutine

Purpose

Initiate/respond to an orderly release with user data.

Library

Syntax

```
#include <xti.h>

int t_sndreldata(int fd, struct t_discon *discon)
```

Description

This function is used to initiate an orderly release of the outgoing direction of data transfer and to send user data with the release. The argument *fd* identifies the local transport endpoint where the connection exists, and **discon** points to a **t_discon** structure containing the following members:

```
struct netbuf udata;
int reason;
int sequence;
```

After calling **t_sndreldata**, the user may not send any more data over the connection. However, a user may continue to receive data if an orderly release indication has not been received.

The field **reason** specifies the reason for the disconnection through a protocol-dependent **reason code**, and **udata** identifies any user data that is sent with the disconnection; the field **sequence** is not used.

The **udata** structure specifies the user data to be sent to the remote user. The amount of user data must not exceed the limits supported by the transport provider, as returned in the **discon** field of the *info* argument of **t_open** or **t_getinfo**. If the **len** field of **udata** is zero or if the provider did not return T_ORDRELDATA in the **t_open** flags, no data will be sent to the remote user.

If a user does not wish to send data and reason code to the remote user, the value of **discon** may be a null pointer.

This function is an optional service of the transport provider, only supported by providers of service type T_COTS_ORD. The flag T_ORDRELDATA in the **info->flag** field returned by **t_open** or **t_getinfo** indicates that the provider supports orderly release user data; when the flag is not set, this function behaves as **t_rcvrel** and no user data is returned.

This function may not be available on all systems.

Parameters	Before call	After call
fd	x	/
discon->	udata.maxlen	/
discon->	udata.len	x
discon->	udata.buf	?(?)
discon->	reason	?
discon->	sequence	/

Valid States

T_DATAXFER, T_INREL

Error Codes

On failure, **t_errno** is set to one of the following:

[TBADDATA]
The amount of user data specified was not within the bounds allowed by the transport provider, or user data was supplied and the provider did not return T_ORDRELDATA in the **t_open** flags.

[TBADF]
The specified file descriptor does not refer to a transport endpoint.

[TFLOW]
O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting the function at this time.

[TLOOK]
An asynchronous event has occurred on this transport endpoint and requires immediate attention.

[TNOTSUPPORT]
Orderly release is not supported by the underlying transport provider.

[TOUTSTATE]
The communications endpoint referenced by **fd** is not in one of the states in which a call to this function is valid.

[TPROTO]
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error (**t_errno**).

[TSYSERR]
A system error has occurred during execution of this function.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **t_errno** is set to indicate an error.

t_sndvudata Subroutine

Purpose

Send a data unit from one or more noncontiguous buffers.

Library

Syntax

```
#include <xti.h>

int t_sndvudata(
    int fd,
    struct t_unitdata *unitdata,
    struct t_iovec *iov,
    unsigned int iovcount)
```

Description

This function is used in connectionless mode to send a data unit to another transport user. The argument *fd* identifies the local transport endpoint through which data will be sent, **iovcount** contains the number of non-contiguous udata buffers and is limited to an implementation-defined value given by T_IOV_MAX, which is at least 16, and **unitdata** points to a **t_unitdata** structure containing the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

If the limit on **iovcount** is exceeded, the function fails with [TBADDATA].

In **unitdata**, **addr** specifies the protocol address of the destination user, and **opt** identifies options that the user wants associated with this request. The *udata* field is not used. The user may choose not to specify what protocol options are associated with the transfer by setting the *len* field of **opt** to zero. In this case, the provider may use default options.

The data to be sent is identified by **iov[0]** through **iov[iovcount-1]**.

> The limit on the total number of bytes available in all buffers passed (that is:
>
> ```
> iov(0).iov_len + . . + iov(iovcount-1).iov_len )
> ```
>
> may be constrained by implementation limits. If no other constraint applies, it will be limited by [INT_MAX]. In practice, the availability of memory to an application is likely to impose a lower limit on the amount of data that can be sent or received using scatter/gather functions.

By default, **t_sndvudata** operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NONBLOCK is set (via **fcntl**, **t_sndvudata** executes in asynchronous mode and will fail under such conditions. The process can arrange to be notified of the clearance of a flow control restriction via the EM interface.

If the amount of data specified in **iov[0]** through **iov[iovcount-1]** exceeds the TSDU size as returned in the *tsdu* field of the *info* argument of **t_open** or is zero and sending of zero octets is not supported by the underlying transport service, a [TBADDATA] error is generated. If **t_sndvudata** is called before the destination user has activated its transport endpoint the data unit may be discarded.

Parameters	Before call	After call
fd	x	/
unitdata->	addr.maxlen	/

Parameters	Before call	After call
unitdata->	addr.len	x
unitdata->	addr.buf	x(x)
unitdata->	opt.maxlen	/
unitdata->	opt.len	x
unitdata->	opt.buf	?(?)
unitdata->	udata.maxlen	/
unitdata->	udata.len	/
unitdata->	udata.buf	/
iov[0].iov_base	x(x)	=(=)
left>iov[0].iov_len	x	=
. . . .		
iov[iovcount-1].iov_base	x(x)	=(=)
iov[iovcount-1].iov_len	x	=
iovcount	x	/

Valid States

T_IDLE

Error Codes

On failure, **t_errno** is set to one of the following:

Item	Description
[TBADADDR]	The specified protocol address was in an incorrect format or contained illegal information.
[TBADDATA]	Illegal amount of data. • A single send was attempted specifying a TSDU greater than that specified in the *info* argument, or a send of a zero byte TSDU is not supported by the provider. • **iovcount** is greater than T_IOV_MAX.
[TBADF]	The specified file descriptor does not refer to a transport endpoint.
[TBADOPT]	The specified options were in an incorrect format or contained illegal information.
[TFLOW]	O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.
[TLOOK]	An asynchronous event has occurred on this transport endpoint.
[TNOTSUPPORT]	This function is not supported by the underlying transport provider.
[TOUTSTATE]	The communications endpoint referenced by **fd** is not in one of the states in which a call to this function is valid.
[TPROTO]	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI error **(t_errno)**.
[TSYSERR]	A system error has occurred during execution of this function.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and **t_errno** is set to indicate an error.

t_sysconf Subroutine

Purpose

Get configurable XTI variables.

Library

Standard library (**libxti.a**)

Syntax

```
#include <xti.h>
```

```
int t_sysconf (    int name)
```

Description

Parameters	Before call	After call
name	X	/

The **t_sysconf** function provides a method for the application to determine the current value of configurable and implementation-dependent XTI limits or options.

The **name** argument represents the XTI system variable to be queried. The following table lists the minimal set of XTI system variables from **xti.h** that can be returned by **t_sysconf**, and the symbolic constants, defined in **xti.h** that are the corresponding values used for **name**.

Variable	Value of Name
T_IOV_MAX	_SC_T_IOV_MAX

Return Values

If **name** is valid, **t_sysconf** returns the value of the requested limit/option (which might be -1) and leaves **t_errno** unchanged. Otherwise, a value of -1 is returned and **t_errno** is set to indicate an error.

Error Codes

On failure, t_errno is set to the following:

Item	Description
TBADFLAG	**name** has an invalid value.

Related Information

The **t_rcvv** (“t_rcvv Subroutine” on page 2185) subroutine, **t_rcvvudata** (“t_rcvvudata Subroutine” on page 2186) subroutine, **t_sndv** (“t_sndv Subroutine” on page 2188) subroutine, **t_sndvudata** (“t_sndvudata Subroutine ” on page 2193) subroutine.

tparm Subroutine

Purpose

Applies parameters (padding) to a terminal capability.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
char *tparm( TermCap,  Parm1, Parm2, . . . Parm9)
char *TermCap;
int  Parm1, Parm2, . . . Parm9;
```

Description

The **tparm** subroutine applies parameters (padding) to a terminal capability.

Note: If the **tparm** subroutine is called with less than 10 paramameters, then the **-D_TPARM_COMPAT** option should be used when compiling the program. Otherwise the compiler gives the following error.

```
1506-098 (E) Missing argument(s)
```

Parameters

Item	Description
Parm#	Specifies the parameters (up to nine) to instantiate.
TermCap	Specifies the terminal capability to apply the parameters to. These terminal capabilities are defined in the **term.h** file.

Return Values

The **tparm** subroutine returns the escape sequence specified by the *TermCap* parameter with the specified parameters applied. After the escape sequence is received, it can be output by a subroutine like the **tputs** (“tputs Subroutine” on page 2197) subroutine.

Examples

1. To save the escape sequence used to home the cursor in the user-defined variable `home_sequence`, enter:

   ```
   home_sequence = tparm(cursor_home);
   ```

2. To save the escape sequence used to move the cursor to the coordinates X=40, Y=18 in the user-defined variable `move_sequence`, enter:

   ```
   move_sequence = tparm(cursor_address, 18, 40);
   ```

tputs Subroutine

Purpose

Outputs a string with padding information.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
#include <term.h>
```

tputs(*String*, *LinesAffected*, *PutcLikeSub***)**
char *****String**;**
int *LinesAffected***;**
int (*PutcLikeSub**) ();**

Description

The **tputs** subroutine outputs a string with padding information applied. String must be a terminfo string variable or the return value from **tparm**, **tgetstr**, **tigetstr**, or **tgoto** subroutines.

Parameters

Item	Description
LinesAffected	Specifies the number of lines affected, or specifies 1 if not applicable.
PutcLikeSub	Specifies a **putchar**-like subroutine through which the characters are passed one at a time.
String	Specifies the string to which to add padding information.

Examples

1. To output the clear screen sequence using the user-defined **putchar**-like subroutine **my_putchar**, enter:

```
int_my_putchar();
tputs(clear_screen, 1 ,my_putchar);
```

2. To output the escape sequence used to move the cursor to the coordinates x=40, y=18 through the user-defined **putchar**-like subroutine `my_putchar`, enter:

```
int_my_putchar();
tputs(tparm(cursor_address, 18, 40), 1, my_putchar);
```

trc_close Subroutine

Purpose

Closes and frees a trace log object.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_close (handle)
trc_log_handle_t handle;
```

Description

The **trc_close** subroutine closes a trace log object. The object must have been opened with the **trc_open** subroutine. If the **TRC_RETAIN_HANDLE** type was specified at open time, the **trc_close** subroutine must be called after a call to the **trc_open** subroutine, regardless of whether the open succeeded or not.

Parameters

Item	Description
handle	Contains the handle returned from a successful call to the **trc_open** subroutine.

Return Values

Upon successful completion, the **trc_close** subroutine returns a 0.

Error Codes

Upon error, the **trc_close** subroutine sets the **errno** global variable and returrns the error from the **fclose** subroutine. In addition, **EINVAL** is returned if handle contains an invalid **trc_log_handle_t** object.

trc_find_first, trc_find_next, or trc_compare Subroutine

Purpose

Finds the first, or next, occurrence of the argument, or compares the current entry with the argument.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_find_first (handle, argp, ret)
trc_log_handle_t handle;
trc_logsearch_t *argp;
trc_read_t *ret;
```

```
int trc_find_next (handle, argp, ret)
trc_log_handle_t handle;
trc_logsearch_t *argp;
trc_read_t *ret;
```

```
int trc_compare (handle, argp)
trc_log_handle_t handle;
trc_logsearch_t *argp;
```

Description

The **trc_find_first** subroutine finds the first occurrence of the trace log entry matching the argument pointed to by the *argp* parameter. The **trc_find_next** subroutine finds the next occurrence of the argument starting from the current position in the log object. If the search argument pointer, *argp*, is NULL, the argument from the previous search is used. Both the **trc_find_first** and **trc_find_next** subroutines return the item found. If the flag field of the handle contains both TRC_MULTI_MERGE and TRC_REMOVE_DUPS, **trc_find_first** and **trc_find_next** will consume any duplicate entries of the current event that exist from other trace sources. The number of entries consumed will be returned in the *trchi_dupcount* or *trcri_dupcount* variable (depending on whether processed or raw data items, respectively, are requested).

The **trc_compare** subroutine is used to check the current entry against the argument. No data is read. It is useful when implementing exit criteria, where you need to find entries according to some criteria, but then check for an exit criteria which is not part of the normal search.

Parameters

Item	Description
handle	Contains the handle returned from a successful call to the **trc_open** subroutine.
argp	Points to the argument list as defined in the **/usr/include/sys/libtrace.a** file. Arguments may be chained together to perform complex searches.
ret	Points to the **trc_read_t** structure to be returned. The **trc_free** subroutine should be used to free data referenced from the **trc_read_t** data type, unless **TRC_LOGLIVE** was specified at open time.

The search argument consists of three parts, the operator, **tls_op**, and the left and right sides.

The operator values can be easily identified, because they have the form TLS_OP_ . . . . Operators are split into two categories, leaf and compound operators. Leaf operators are operators that compare the field on the left with the value on the right. Compound operators are used to compare two expressions, (for example) to combined expressions.

Leaf operations may be performed using numeric or string data. If performed on string data, the **strcmp libc** string compare function is used to do the comparison for all operators except **TLS_OP_SUBSTR**. The valid leaf operators are:

Item	Description
TLS_OP_EQUAL	Exactly equal
TLS_OP_NE	Not equal
TLS_OP_LT	Less than
TLS_OP_LE	Less than or equal
TLS_OP_GT	Greater than
TLS_OP_GE	Greater than or equal
TLS_OP_SUBSTR	The string on the left contains the string on the right.

The compound operators are:

Item	Description
TLS_OP_AND	The logical AND of the results of the left and right expressions.

Item	Description
TLS_OP_OR	The logical OR of the results of the left and right expressions.
TLS_OP_XOR	The exclusive or of the results of the left and right expressions.
TLS_OP_NOT	The negation of the argument referenced by **tls_left**.

The left and right sides of the expression are defined as follows:

Item	Description
tls_left and **tls_right**	These are used when the operator requires the left and right sides to be an expression, (for example) when it is a compound operator. **tls_left** and **tls_right** point to other **trc_logsearch_t** structures.
tls_field and corresponding values	For a leaf operation, **tls_field**, on the left, specifies the field to be compared. The field names can be identified easily, because they all have the form `TLS_MATCH_...`. The righthand side is a value specified according to the data type of the field on the left.

The following table shows the lefthand field values and their corresponding righthand side data values:

Field	Value	Description
TLS_MATCH_HOOKID	**tls_ushortvalue**	Compare the hook ID with a **ushort** data item. Only a 3-digit hook ID can be used. Beginning with AIX 6.1 where 4-digit hook IDs are available, arguments are left-shifted by 4 to create a 4-digit hook ID. For example, to specify the hook ID 0x1000, specify 0x100. To specify the hook ID 0x00F0, specify 0x00F. Thus, only 4-digit hook IDs in the form of 0x*hhh*0 can be specified where *h* is a hexadecimal digit. To specify any 4-digit hook ID, use **TLS_MATCH_HOOKID64**.
TLS_MATCH_HOOKID64	**tls_ushortvalue**	Valid beginning with AIX 6.1. Compare the hook ID with a **ushort** data item. All hook IDs are assumed to be 4-digit hook IDs.

Field	Value	Description
TLS_MATCH_HOOK_AND_SUBHOOK	**tls_uintvalue**	Compare the hook and subhook. Use 32 bits with the specified integer. The field is in the form of 0x*hhhhssss*, where *hhhh* is the hook ID (and can optionally be *hhh0*), and *ssss* is the subhook.
TLS_MATCH_HOOKSET	**tls_hooksetvalue**	The bitmap specifies the hooks to be tested for. You can test for multiple hooks with one search argument. The bit map is manipulated with the **trc_hkemptyset**, **trc_hkfillset**, **trc_hkaddset**, and **trc_hkdelset** subroutines. Beginning with AIX 6.1, 16-bit hook IDs are available. However, **trc_hookset_t** can only specify 12-bit hook IDs. By specifying hook 0x*hhh*, the **trc_find_first** and **trc_find_next** subroutines search for 0x*hhh*0 because the two values are equivalent beginning with AIX 6.1. To specify hooks in the form of 0x*hhhh*, use **TLS_MATCH_HOOKSET64** .
TLS_MATCH_HOOKSET64	**tls_hooksetvalue**	Valid beginning with AIX 6.1. The bitmap specifies the hook IDs to be tested for. You can test for multiple 16-bit hooks with one search argument. The bit map is manipulated with the **trc_hkemptyset64**, **trc_hkfillset64**, **trc_hkaddset64**, and **trc_hkdelset64** subroutines.
TLS_MATCH_TIME	**tls_ulongvalue**	Compare the time value in nanoseconds from the start of the trace.
TLS_MATCH_TID	**tls_ulongvalue**	Thread ID
TLS_MATCH_PID	**tls_ulongvalue**	Process ID
TLS_MATCH_RAWOFST	**tls_ulongvalue**	Raw file offset

Field	Value	Description
TLS_MATCH_CPUID	**tls_uintvalue**	Processor ID
TLS_MATCH_RCPU	**tls_uintvalue**	Remaining processors in the trace
TLS_MATCH_FLAGS	**tls_uintvalue**	Compare with **trcr_flags**.
TLS_MATCH_TICKS	**tls_ulongvalue**	Match with the number of timer register ticks since the start of the trace.
TLS_MATCH_TRCONTIME	**tls_ulongvalue**	Compare with **trchi_trcontime**.
TLS_MATCH_TRCOFFTIME	**tls_ulongvalue**	Compare with **trchi_trcofftime**.
TLS_MATCH_COMPONENT	**tls_strvalue**	Match a specific component name.

Return Values

Upon successful completion, the **trc_find_first**, **trc_find_next**, and **trc_compare** subroutines return 0.

Error Codes

Upon error, the **errno** global variable is set to a value from the **errno.h** file. The **trc_find_first**, **trc_find_next**, and **trc_compare** subroutines return either a value from the **errno.h** file, or an error value from the **libtrace.h** file.

Item	Description
EINVAL	The handle is invalid, or the search argument is invalid.
TRCE_EOF	No matching item was found, or no more matching items exist. The **errno** global variable is set to 0.
TRCE_BADFORMAT	The log object contains badly formatted data. The **errno** global variable is set to **EINVAL**.

Examples

1. Find the SVC hooks, 101 and 104, for program `mypgm`.

```
{
                    int rv;
                    trc_loghandle_t h;
                    trc_read_t r;
                    trc_logsearch_t t1, t2, t3, t4, t5;

                    /* Setup the leaf search arguments. */
                    t1.tls_op = TLS_OP_EQUAL;
                    t1.tls_field = TLS_MATCH_HOOKID;
                    t1.tls_ushortvalue = 0x101;
                    t2.tls_op = TLS_OP_EQUAL;
                    t2.tls_field = TLS_MATCH_HOOKID;
                    t2.tls_ushortvalue = 0x104;
                    t3.tls_op = TLS_OP_EQUAL;
                    t3.tls_field = TLS_MATCH_PROCNAME;
                    t3.tls_strvalue = "mypgm";
                    /* Join the items and form a single search tree. */
                    t4.tls_op = TLS_OP_AND;
                    t4.tls_left = &t1
                    t4.tls_right = &t2
                    t5.tls_op = TLS_OP_AND;
                    t5.tls_left = &t4
```

```
                t5.tls_right = &t3
                /* Open the default trace log object. */
                rv = trc_open("", "", TRC_LOGREAD|TRC_LOGPROC, >h);
                if (rv) {
                        trc_perror(h, rv, "open");
                        return(rv);
                }
                /* Do the search. */
                rv = trc_find_first(h, &t5, &r);
                if (rv) {
                        trc_perror(h, rv, "find test");
                        return(rv);
                }

                ...
        }
```

Note that subsequent entries matching this search could be returned with the following:

```
rv = trc_find_next(h, NULL, &r);
```

After a find, **trc_find_next** can be used to change the search argument without starting the search over. In other words, **trc_find_first** always starts from the beginning of the file, while **trc_find_next** starts from the current position in the file, but either one can change the search argument.

2. Find the SVC hooks, 101 and 104, for program `mypgm`. Use a single argument to search for both hook ids.

```
{
                int rv;
                trc_loghandle_t h;
                trc_read_t r;
                trc_logsearch_t t1, t2, t3;
                trc_hookset_t hs;

                /* Setup the hook set. */
                trc_hkemptyset(hs);
                (void)trc_hkaddset(hs, 0x101);
                (void)trc_hkaddset(hs, 0x104);
                /* Setup the leaf search arguments. */
                t1.tls_op = TLS_OP_EQUAL;
                t1.tls_field = TLS_MATCH_HOOKSET;
                t1.tls_hooksetvalue = hs;
                t2.tls_op = TLS_OP_EQUAL;
                t2.tls_field = TLS_MATCH_PROCNAME;
                t2.tls_strvalue = "mypgm";
                /* Join the items and form a single search tree. */
                t3.tls_op = TLS_OP_AND;
                t3.tls_left = &t1
                t3.tls_right = &t2
                /* Open the default trace log object. */
                rv = trc_open("", "", TRC_LOGREAD|TRC_LOGPROC, &h);
                if (rv) {
                        trc_perror(h, rv, "open");
                        return(rv);
                }
                /* Do the search. */
                rv = trc_find_first(h, &t3, &r);
                if (rv) {
                        trc_perror(h, rv, "find test");
                        return(rv);
                }

                ...
        }
```

3. You can find hooks 101, 104 and 1AB1 for program **mypgm** using the **trc_hookset64_t** type. Hooks 101 and 104 are equal to 0x1010 and 0x1040.

```
{
                int rv;
                trc_loghandle_t h;
                trc_read_t r;
                trc_logsearch_t t1, t2, t3;
                trc_hookset64_t hs;

                /* Setup the hook set. */
```

```
                trc_hkemptyset64(hs);
                (void)trc_hkaddset64(hs, 0x1010);
                (void)trc_hkaddset64(hs, 0x1040);
                (void)trc_hkaddset64(hs, 0x1AB1);

                /* Setup the leaf search arguments. */
                t1.tls_op = TLS_OP_EQUAL;
                t1.tls_field = TLS_MATCH_HOOKSET64;
                t1.tls_hooksetvalue = hs;
                t2.tls_op = TLS_OP_EQUAL;
                t2.tls_field = TLS_MATCH_PROCNAME;
                t2.tls_strvalue = "mypgm";
                /* Join the items and form a single search tree. */
                t3.tls_op = TLS_OP_AND;
                t3.tls_left = &t1
                t3.tls_right = &t2
                /* Open the default trace log object. */
                rv = trc_open("", "", TRC_LOGREAD|TRC_LOGPROC, &h);
                if (rv) {
                        trc_perror(h, rv, "open");
                        return(rv);
                }
                /* Do the search. */
                rv = trc_find_first(h, &t3, &r);
                if (rv) {
                        trc_perror(h, rv, "find test");
                        return(rv);
                }

                . . .
        }
```

trc_free Subroutine

Purpose

Frees memory allocated by the **trc_read**, **trc_find**, **trc_loginfo**, or **trc_hookname** subroutine.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_free (parmp)
void *parmp;
```

Description

The **trc_free** subroutine is used to free memory associated with data structures returned by the trace retrieval API. It does not free the storage for the base structure, however, only storage allocated by the API on behalf of the user. The pointer must point to one of the following:

trc_read_t
Data returned by the **trc_read** or **trc_find** subroutine.

trc_loginfo_t
Data returned by the **trc_loginfo** subroutine.

trc_hookname_t
Data returned by the **trc_hookname** subroutine.

trc_logpos_t
A log position object returned by the **trc_tell** subroutine.

A log handle, **trc_loghandle_t**, must be freed using the **trc_close** subroutine.

For example, `trc_free(&trc_data)`, where *trc_data* is of type **trc_read_t**, frees the storage referenced by the **trc_data** structure, but does not free **trc_data** since it must be pre-allocated by the user.

Parameters

Item	Description
parmp	Points to a structure as described above.

Return Values

Upon successful completion, the **trc_free** subroutine returns 0.

Error Codes

Item	Description
EINVAL	The *parmp* parameter points to an unsupported data type.

trc_hkemptyset, trc_hkfillset, trc_hkaddset, trc_hkdelset, or trc_hkisset Subroutine

Purpose

Manipulates a trace hook set of the **trc_hookset_t** type.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

void trc_hkemptyset(hookset)
trc_hookset_t hookset;

void trc_hkfillset(hookset)
trc_hookset_t hookset;

int trc_hkaddset(hookset, hook)
trc_hookset_t hookset;
short hook;

int trc_hkdelset(hookset, hook)
trc_hookset_t hookset;
short hook;

int trc_hkisset (hookset, hook)
trc_hookset_t hookset;
short hook
```

Description

These subroutines manipulate a trace hook set used by the **trc_find** subroutine before AIX 6.1. This hook set can be used to search for several trace hooks simultaneously.

Beginning with AIX 6.1, which supports 16-bit hook IDs, the **trc_hkemptyset**, **trc_hkfillset**, **trc_hkaddset**, **trc_hkdelset**, and **trc_hkisset** subroutines can only operate on 16-bit hook IDs in the form of 0x*hhh*0 where *h* is a hexadecimal digit. Hook IDs in the form of 0x*hhh*0 are equivalent to 12-bit hook IDs in the form of 0x*hhh* before AIX 6.1 . To work with the entire expanded hook ID range beginning

with AIX 6.1, use the **trc_hookset64_t** type and its manipulation subroutines (the **trc_hkemptyset64**, **trc_hkfillset64**, **trc_hkaddset64**, **trc_hkdelset64**, and **trc_hkisset64** subroutines).

Parameters

Item	Description
hookset	References the hook set to be operated on.
hook	Specifies a hook value in the range 0x000 - 0xfff.

Return Values

The **trc_hkaddset**, **trc_hkdelset**, and **trc_hkisset** subroutines return **EINVAL** if the hook is out of range (that is, greater than 0xfff).

The **trc_hkaddset** subroutine returns 0 if the hook wasn't in the set, and -1 if it was already present.

The **trc_hkdelset** subroutine returns 0 if the hook was in the set, and -1 if it wasn't present.

The **trc_hkisset** subroutine returns 0 if the hook isn't present, and -1 if it is present.

trc_hkemptyset64, trc_hkfillset64, trc_hkaddset64, trc_hkdelset64, or trc_hkisset64 Subroutine

Purpose

Manipulates a trace hook set of the **trc_hookset64_t** type.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>
```

```
void trc_hkemptyset64(hookset)
trc_hookset64_t hookset;

void trc_hkfillset64(hookset)
trc_hookset64_t hookset;

int trc_hkaddset64(hookset, hook)
trc_hookset64_t hookset;
short hook;

int trc_hkdelset64(hookset, hook)
trc_hookset64_t hookset;
short hook;

int trc_hkisset64(hookset, hook)
trc_hookset64_t hookset;
short hook;
```

Description

The **trc_hkemptyset64**, **trc_hkfillset64**, **trc_hkaddset64**, **trc_hkdelset64**, and **trc_hkisset64** subroutines manipulate the trace hook set used by “trc_find_first, trc_find_next, or trc_compare Subroutine” on page 2198. The hook set can be used to search for several trace hooks simultaneously. The **trc_hkfillset64** subroutine sets all hook IDs except for 0x0000 and hook IDs less than 0x1000 where the least significant digit is not 0 (for example, 0x0hh1 is not valid).

Parameters

Item	Description
hookset	References the hook set to be operated on.
hook	Specifies a hook value ranging from 0x0000 through 0xffff.

Return Values

Item	Description
trc_hkaddset64	• EINVAL – The hook is not valid (0 or less than 0x1000 with a nonzero value in the least significant digit). • 0 – The hook is in the set. • -1 – The hook is not present.

Item	Description
trc_hkdelset64	• EINVAL – The hook is not valid (0 or less than 0x1000 with a nonzero value in the least significant digit). • 0 – The hook is in the set. • -1 – The hook is not present.

Item	Description
trc_hkisset64	• EINVAL – The hook is not valid (0 or less than 0x1000 with a nonzero value in the least significant digit). • 0 – The hook is in the set. • -1 – The hook is not present.

trc_hookname Subroutine

Purpose

Returns one or all hooks and associated names from the template file.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_hookname (handle, hook, hooknamep)
trc_log_handle_t handle;
trc_hookid_t hook;
trc_hookname_t *hooknamep;
```

Description

The **trc_hookname** subroutine returns one or more hook ids and their associated descriptions. This allows a trace data formatter to provide a hook selection list with some descriptive text for each hook.

Parameters

Item	Description
handle	Contains a **trc_log_handle_t** data item returned from a successful call to the **trc_open** subroutine.
hook	Before AIX 6.1, the *hook* parameter contained a hook ID in the form of 0x*hhh* where *hhh* was a 3-hex-digit hook ID. Beginning with AIX 6.1, the *hook* parameter contains a hook ID in the form of 0x*hhhh* where *hhhh* is a 4-hex-digit hook ID. If the *hook* parameter is **TRC_HOOK_ALL**, the names for all of the hooks in the template file are returned.
hooknamep	Points to a **trc_hookname_t** structure. The **trc_free** subroutine should be used to free any data referenced by the **trc_hookname_t** data item.

```
/* Array element type for hook ids and names. */
     typedef struct {
             trc_hookid_t hookid;
             char *hookname;
     } trc_hooknm_t;

     typedef struct {
             int trchn_magic;                /* Identifier for this data structure. */
             unsigned trchn_nhooks;          /* Number of hooks. */
             trc_hooknm_t *trchn_names;      /* Pointer to array of ids and names. */
     } trc_hookname_t;
```

Return Values

Upon successful completion, the **trc_hookname** subroutine returns 0.

Error Codes

Item	Description
ENOMEM	Not enough memory to satisfy the request.
TRCE_WARN	A formatting error was found in the template file. If **TRCE_WARN** is returned, the function completed.
TRCE_BADFORMAT	A formatting error was found in the template file. If **TRCE_BADFORMAT** was returned, the **errno** global variable is set to **EINVAL**.

trc_ishookon Subroutine

Purpose

Check if a given trace hook word is being traced by system trace.

Library

Runtime Services Library (**librts.a**)

Syntax

```
#include <sys/trcmacros.h>

int trc_ishookon(int chan, long hkwd)
```

Description

The **trc_ishookon** subroutine returns 1 if tracing for the specified channel is on and the specified hook word is being traced, otherwise it returns 0.

Parameters

Item	Description
chan	The channel to query ranging from channel number 0 though 7.
hkwd	The hook word to be traced by system trace.

Return Values

Item	Description
1	The specified hook word is being traced.
0	Hook word is not being traced or system trace is off.

Files

/dev/systrct1[-{0-7}]

trc_ishookset Subroutine

Purpose

Return an indication of all hooks currently being traced.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_ishookset(int chan, char *hkst, size_t hkst_sz)
```

Description

The **trc_ishookset** subroutine returns 1 if the specified channel is being traced, 0 otherwise. If it returns 1, the hookset item is modified to contain an indication of the hooks being traced. The facilities in the **libtrace.a** library for examining a data item of **trc_hookset_t** or **trc_hookset64_t** type can then be used.

If data of the **trc_hookset_t** type is passed on a system before AIX 6.1, the status of all 12-bit hook IDs are returned. If data of the **trc_hookset_t** type is passed on AIX 6.1 and later, only information about the hooks of the form 0x*hhh*0 (represented as 0x*hhh*) is returned where *h* is a hexadecimal digit. If data of the **trc_hookset64_t** type, which is valid beginning with AIX 6.1, is passed, information about all 16-bit hook IDs is returned.

Parameters

Item	Description
chan	The channel to query ranging from channel number 0 through 7.

Item	Description
hkst	Pointer to a variable of type **trc_hookset_t** or **trc_hookset64_t**.
hkst_sz	Size of the hookset being passed in.

Return Values

Item	Description
1	System trace is on.
0	System trace is off.

Files

/dev/systrct1[-{0-7}]

trc_libcntl Subroutine

Purpose

Performs trace API control functions.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_libcntl (handle, cmd, datap)
trc_log_handle_t handle;
int cmd;
void *datap;
```

Description

The **trc_libcntl** subroutine provides miscellaneous control functions.

Parameters

Item	Description
handle	Contains the handle returned from a successful call to the **trc_open** subroutine.

Item	Description
cmd	This is the control function to be performed. Supported functions are: **TRC_CNTL_ADJLINENO** This allows a trace report program to adjust the **$LINENO** value supplied through the trace templates. Normally, a trace reporting program may assume the **$LINENO** value is calculated based upon the first line of the output, in **trchi_ascii**, being the first line printed for that hook in the report. If this is not the case, such as with the **2line trcrpt** option, the **$LINENO** value must be adjusted. For **TRC_CNTL_ADJLINENO**, the *datap* parameter must contain a signed long value which is added to **$LINENO**. If the value is negative, **TRC_CNTL_ADJLINENO** will decrement the value. **TRC_CNTL_NAMELIST** This allows the namelist to be specified. The default is **/unix**. It does not initialize the symbols, however, and the **trc_libcntl** subroutine returns **EINVAL** if the symbols are already initialized. If symbols are in the trace stream, specified by `trace -n`, those symbols are used regardless of the namelist specification. **TRC_CNTL_TEXTOFFSET** This offsets each line of text, in the **trchi_ascii** data area, by the number of character positions specified, plus `(trchi_indent-1) * 8;` If the associated value is 0, each line is only offset by `(trchi_indent-1) * 8;` **TRC_CNTL_TEXTOFFSET_SUBSEQUENT** This works exactly like **TRC_CNTL_TEXTOFFSET**, except it offsets all lines except the first line of text. The first line is still offset by `(trchi_indent-1) * 8;` **TRC_CNTL_PAGESIZE** This specifies the length of a page. **TRC_CNTL_TEXTHEADER** This specifies a header to be output every page, as specified by the **TRC_CNTL_PAGESIZE** command.
datap	Specifies the data parameter.

Return Values

Upon successful completion, the **trc_libcntl** subroutine returns 0.

Error Codes

Item	Description
EINVAL	The *handle* or *cmd* parameter is invalid. **EINVAL** is also returned if the value specified with **TRC_CNTL_ADJLINENO** would cause the **$LINENO** value to be negative.

trc_loginfo Subroutine

Purpose

Returns information about a trace log object.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_loginfo (log_object_name, infop)
char *log_object_name;
trc_log_info_t *infop;
```

Description

The **trc_loginfo** subroutine returns information about the named trace log object. If the *log_object_name* parameter is NULL or an empty string, the **trc_loginfo** subroutine returns information about the default log object.

Parameters

Item	Description
log_object_name	Names the trace log object. This is specified as it is for the **trc_open** subroutine.
infop	Points to an item of type **trc_log_info_t** where the information will be returned. The **trc_log_info_t** structure is defined in the **/usr/include/sys/libtrace.h** file. It contains such fields as the file size, the time the trace was taken, the trace log file magic number, the command used to start the trace, CPUs in the machine, number of CPUs traced, multi-CPU trace indicator (**-C**), and the trace object type as defined in the **trcopen** subroutine. The **trc_free** subroutine should be called to free the **trc_loginfo_t** information, even if the **trc_loginfo** subroutine returned an error.

The **/usr/include/sys/libtrace.h** file contains the data definitions for the returned data, ***infop**. The following table contains the data item name, data type, and description for each item returned:

Label	Data Type	Description
trci_magic	**int**	Structure magic number managed by the library.
trci_logmagic	**int**	The trace log file's magic number, see the **/usr/include/sys/trchdr.h** file. This identifies the type of log file, and is included mainly for completeness. The pertinent log file information may be gotten from other fields in this structure.
trci_time	**time_t**	The time the trace was taken.
trci_ipaddr	**int**	The system's IP address.
trci_uname	**struct utsname**	uname information.
trci_cmd	**char ***	The command used to start the trace.
trci_fnames	**trci_fname_t***	Log file names array.
trci_mach_cpus	**int**	Number of CPUs in the machine.
trci_traced_cpus	**int**	Number of traced CPUs.
trci_flags	**int**	Data stream flags.
trci_obj_type	**int**	Trace object type.

Label	Data Type	Description
trci_hookids	**trc_hookset_t** or **trc_hookset64_t**	The binary hook IDs map shows the hooks traced. If the application is compiled on systems before AIX 6.1, the **trc_hookset_t** data type is provided and can be examined with the **trc_hkisset** subroutine. Beginning with AIX 6.1, applications are provided with the **trc_hookset64_t** type that can be examined with the **trc_hkisset64** subroutine.

The **trci_flags** field contains bit flags as follows:

Item	Description
TRCIF_MULTICPU	This trace was taken with the **-C** trace option, (for example) it is a multi-CPU trace.
TRCIF_64BIT	This is a 64-bit trace, 32-bit if not set.
TRCIF_SEPSEG	Separate segment buffering was used.
TRCIF_CONDTRACE	Conditional trace by hookid, trace **-j**, **-k**, **-J**, or **-K**.
TRCIF_CONDEXCL	Trace hook exclusion, **-k** or **-K**, was used.
TRCIF_COMPONENT	The given file is a Component Trace master file obtained by either the **ctctrl** command or the **trcdead** command.

Return Values

Upon successful completion, the **trc_loginfo** subroutine returns a 0, and information about the trace log object is placed into the memory pointed to by the *infop* parameter.

Error Codes

Upon error, the **trc_loginfo** subroutine returns information identical to that returned by the “trc_open Subroutine” on page 2214.

trc_logpath Subroutine

Purpose

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

            char *trc_logpath(void)
```

Description

The **trc_logpath** subroutine returns the default trace logfile path name. This is normally **/var/adm/ras/trcfile**, unless changed with the **trcctl** command or SMIT. Any process that can access and link to the **libtrace.a** library can call the **trc_logpath** subroutine and retrieve the current path to the default trace file. With the addition of the **trcctl** command to the available administration options, system administrators can now set the default to any path rather than always having **/var/adm/ras/trcfile** as the hard-coded default. Trace Report **trcrpt** calls the library routines **trc_open** and **trc_loginfo** to access the trace file. Beginning with AIX 5.3, **trc_open** and **trc_loginfo** both call **trc_logpath** to access the default file, if it is required. Calling **trc_logpath** is transparent to **trcrpt** and the Trace GUI; however, because **trc_logpath** is available and exported in **libtrace.a**, other components and third-party products can use it.

Return Values

The **trc_logpath** subroutine always returns a path name. The path name should be freed, **free**(*path*), by the user when appropriate.

trc_open Subroutine

Purpose

Opens a trace log object.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_open (log_object_name, template_file_name, type, handlep)
char *log_object_name, template_file_name;
int type;
trc_log_handle_t *handlep;
```

Description

The **trc_open** subroutine opens a trace log object. A log object may only be opened for reading.

Two object types are supported, raw and processed. As their names imply, a raw object consists of the raw trace data as it was traced. A processed object consists of data as processed by a trace formatting template file such as the **/etc/trcfmt** file.

Parameters

Item	Description
log_object_name	Specifies the log object to be opened. If this is NULL or an empty string, the default log object, **/var/adm/ras/trcfile**, is opened. If it is a dash, the input is read from standard input. In this case, the file must be a sequential trace file such as one produced by the **trcrpt -r** command, the **-o** trace option, or the **trcdead** command. If the file is the base file for a multi-CPU trace, the trace events are merged by the **trcrpt** command, unless the **TRC_NOTEMPLATES** option was specified. Also, if the file is a single CPU's trace file, it is treated as a single log file. If multiple files are specified for merging, the **TRC_MULTI_MERGE** option must be specified. Each file must be separated from the previous one by a colon. For example, merging 3 files (f1, f2 and f3) is accomplished by setting the *log_object_name* parameter to `f1:f2:f3`.
template_file_name	This names the template file. The template file is used if the **TRC_LOGPROC** type is specified. If NULL, **/etc/trcfmt** (the default template file) is used. The template file specification is ignored if the **TRC_NOTEMPLATES** option is specified.

Item	Description
type	Consists of flag bits OR'd together. One open type and one object type flag must be specified. The following is the open type flag: **TRC_LOGREAD** Open for reading The following are the object type flags: **TRC_LOGRAW** Specifies that raw trace data is to be read. This data is defined in Debug and Performance Tracing and in the **/etc/trcfmt** file. **TRC_LOGPROC** This processes a raw trace log file, one produced by the **trace** command, using either the trace templates found in the **/etc/trcfmt** file, or the template file specified by the *template_file_name* parameter on the **trc_open** command. The following are the modifier type flags: **TRC_LOGVERBATIM** Returns the file data verbatim, exactly as traced. This is how **trcrpt -r** returns data. See also the **TRC_NOTEMPLATES** modifier. **TRC_LIBDEBUG** Turns on debug mode. This is for IBM customer support use only. **TRC_LOGLIVE** The data returned in the **trc_read_t** structure is not a unique copy, it is live data. Such data may only be used until the next retrieval API operation. It is not necessary to call the **trc_free** subroutine to free such data. The **TRC_LOGLIVE** modifier is used to improve performance when the data read does not need to be retained. **TRC_RETAIN_HANDLE** Don't free the handle after an open failure. This allows errors to be processed by the **trc_perror** or **trc_strerror** subroutines. The **trc_close** subroutine must be used to free the file handle. **TRC_NOTEMPLATES** Ignore any template file. This is used with the **TRC_LOGRAW** object flag to prevent any template processing, such as merging multi-CPU trace files. When used in conjunction with the **TRC_LOGVERBATIM** flag, it causes the retrieval API to return the same data reported with **trcrpt -r**. **TRC_MULTI_MERGE** Perform a merge operation on the files specified. Multiple files must be specified. **TRC_REMOVE_DUPS** If set, duplicate entries are eliminated when possible. Duplicate entries can only be detected when the CPU ID is known from the trace entry itself, not when it must be inferred. You can find out what the CPU ID is from the following trace sources: • A lightweight memory trace • A multi-processor system trace (For example, use **trace -C all**.) • A 64-bit system trace initiated with the **-p** option • A 64-bit component trace This flag is valid only when **TRC_MULTI_MERGE** is specified.

Item	Description
handlep	Points to the handle returned from a successful call to the **trc_open** subroutine.

Return Values

Upon successful completion, the **trc_open** subroutine returns a 0 and puts the trace log object handle into the memory pointed to by the *handlep* parameter.

Error Codes

Upon error, the **trc_open** subroutine sets the **errno** global variable to a value in the **errno.h** file, and returns either an **errno.h** value, or an error value defined in the **libtrace.h** file.

Item	Description
EINVAL	Invalid parameter.
ENOMEM	Cannot allocate memory.
TRCE_BADFORMAT	The file is not a valid trace file, and **errno** is set to **EINVAL**.
TRCE_WARN	The template file contains errors. The **errno** global variable is set to **EINVAL** if **TRCE_TMPLTFORMAT** is returned. If **TRCE_WARN** is returned, the open succeeded.
TRCE_TMPLTFORMAT	The template file contains errors. The **errno** global variable is set to **EINVAL** if **TRCE_TMPLTFORMAT** is returned. If **TRCE_WARN** is returned, the open succeeded.
TRCE_TOOMANY	An internal limit is exceeded. The **errno** global variable is set to **ENOMEM** in this case.

trc_perror Subroutine

Purpose

Prints all errors associated with a trace log object.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

void trc_perror (handle, rv, str)
void *handle;
int rv;
char *str;
```

Description

The **trc_perror** subroutine works like the **perror** subroutine. If the error in the *rv* parameter is an error from the **errno.h** file, it behaves exactly like the **perror** subroutine.

If there are multiple errors associated with the handle, the **trc_perror** subroutine prints all errors associated with the object. If the *str* parameter is NULL, the error's text is the only text printed. Errors are printed to standard error.

Parameters

Item	Description
handle	Contains the handle returned from the call to the **trc_open** subroutine, the **trc_logpos_t** object returned by the call to the **trc_loginfo** subroutine,or NULL. If a handle returned by the **trc_open** subroutine is passed, the **trc_open** subroutine need not have been successful, and the **TRC_RETAIN_HANDLE** option must have been used.
rv	The return value from a **libtrace** subroutine.
str	Used the same as the string passed to the **perror** subroutine. Errors printed by the **trc_perror** subroutine are printed as `str: error-message`.

trc_read Subroutine

Purpose

Reads from a trace log object.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_read (handle, ret)
trc_log_handle_t handle;
trc_read_t *ret;
```

Description

The **trc_read** subroutine reads the next sequential data item from the trace log object whose handle is contained in the *handle* parameter. If the **trc_read** subroutine follows a **trc_find_first** or **trc_find_next** call, it reads the next sequential data item after the one found. To read the next item matching that criteria, use the **trc_find_next** subroutine. If the *handle* flag field contains both **TRC_MULTI_MERGE** and **TRC_REMOVE_DUPS**, the **trc_read** subroutine consumes any duplicate entries of the current event that might exist from other trace sources. The number of entries consumed will be returned in the **trchi_dupcount** or **trcri_dupcount** variable (depending on whether processed or raw data items, respectively, are requested) described in the Parameters section.

Parameters

Item	Description
handle	Contains the handle returned from a successful call to the **trc_open** subroutine.

Item	Description
ret	Points to the **trc_read_t** structure to contain the returned information. The raw data will be formatted the same way it is formatted today in the **trcrpt** internal data buffer. This is described in the **/etc/trcfmt** file for both 32 and 64 bit events. Thus 32-bit trace items will be formatted as 32-bit items regardless of whether they came from a 32 or 64 bit trace. If **TRC_LOGVERBATIM** was specified, data is returned exactly as traced. Processed data is the result of trace template processing, see the **/etc/trcfmt** file. The **trc_free** subroutine should be used to free data referenced from the **trc_read_t** data type. The **trc_free** subroutine need not be used if the **TRC_LOGLIVE** flag was specified when the object was opened. The **/usr/include/sys/libtrace.h** file contains the data definitions for the returned data.

The following are definitions for the **trc_read_t** structure. They are split into three sections:

- Definitions for both raw and processed data items
- Definitions for raw data items only
- Definitions for processed data items only

Label	Data Type	Description
trcr_magic	**int**	Trace read data magic number. This is maintained by the library to identify the library version in use.
trcr_flags	**int**	Flags that describe the data returned.

The following are definitions for raw data items:

Label	Data Type	Description
trcri_hookid	**trc_hookid_t**	If the trace entry comes from a 32-bit source, the hook ID is in the form of 0x0*hhh*, where *hhh* is a 3-hex-digit hook ID value (for example, 134). If the trace entry comes from a 64-bit source, the hook ID is in the form of 0x0*hhh* before AIX 6.1. Beginning with AIX 6.1, 16-bit hook IDs are available for 64-bit sources. 16-bit hook IDs in the form of 0x*hhh*0 (for example, 0x1340) are represented as 0x0*hhh* (0x0134) while 16-bit hook IDs in the form of 0x*hhhh* have the value of 0x*hhhh*.
trcri_subhookid	**trc_subhookid_t**	Subhook ID.
trcri_cpuid	**unsigned**	The CPU ID if known. If the **TRCRF_CPUIDOK** flag is set, the CPU ID value could be determined, otherwise it should be ignored.

Label	Data Type	Description
trcri_tid	**unsigned long long**	Thread ID.
trcri_timestamp	**unsigned long long**	Specifies the timestamp in ticks. Use the **trc_ticks2nanos** function to convert this value to nanoseconds.
trcri_rawofst	**unsigned long long**	The offset to the start of this trace item in the trace log file.
trcri_rawlen	**int**	The length of the raw data as traced. This is not necessarily the amount of space used for the data in the log file.
trcri_rawbuf	**char ***	Pointer to the raw data.
trcri_component	**char ***	Current component name. Valid only when processing a component trace log file.
trcri_logfile	**char ***	Current file name.
trcri_dupcount	**int**	Number of events consumed by this **trc_read** call.

TRC_LONGD1(*r*) - **TRC_LONGD5(*r*)** return the 5 data words traced by non-generic trace hooks. The *r* value is of type **trc_read_t ***, and must point to a **trc_read_t** item. These macros return unsigned, 64-bit values.

Note: These macros do not check to ensure that the specified register was traced.

The following are definitions for processed data items:

Label	Data Type	Description
trchi_hookid	**trc_hookid_t**	If the trace entry came from a 32-bit source, the hook ID is in the form of 0x0*hhh*, where *hhh* is a 3-hex-digit hook ID value (for example, 134). If the trace entry comes from a 64-bit source, the hook ID is in the form of 0x0*hhh* before AIX 6.1. Beginning with AIX 6.1, 16-bit hook IDs are available to 64-bit sources. 16-bit hook IDs in the form of 0x*hhh*0 (for example, 0x1340) are represented as 0x0*hhh* (0x0134) while 16-bit hook IDs in the form of 0x*hhhh* have the value of 0x*hhhh*.
trchi_subhookid	**trc_subhookid_t**	Subhook ID.
trchi_elapsed_nseconds	**unsigned long long**	The elapsed time from the start of the trace in nanoseconds.
trchi_tid	**unsigned long long**	Thread ID.
trchi_pid	**unsigned long long**	Process ID.

Label	Data Type	Description
trchi_svc	**unsigned long long**	System call address.
trchi_rawofst	**unsigned long long**	Offset of the trace event in the log file.
trchi_trcontime	**time64_t**	The time of the last **TRCON**, or this **TRCON**.
trchi_trcofftime	**time64_t**	The time of the last **TRCOFF**, or this **TRCOFF**.
trchi_cpuid	**int**	CPU ID.
trchi_rcpu	**int**	CPUs remaining in this trace.
trchi_pri	**int**	Process priority.
trchi_intr_depth	**int**	Interrupt depth.
trchi_indent	**int**	The indentation level used by **trcrpt**. The values are -1 - **$NOPRINT**, 0 - no indentation, 1 - application level, 2 - SVC level, 3 - kernel level. Items greater than zero specify the number of tabs, minus 1, that precede each line of the ascii data, see the **trchi_ascii** field. Each tab represents 8 blanks, so `trchi_indent = 2` implies 2 - 1, or 1 tab before each line of data, or 8 blanks.
trchi_svcname	**char ***	Current svc name.
trchi_procname	**char ***	Current process name.
trchi_filename	**char ***	Current file name.
trchi_ascii	**char ***	This is the data produced by the trace template for this hook. Each line of data is indented with blanks, according to the **trchi_indent** value, and the text offset and the subsequent line offset, see the **trc_libcntl** subroutine.
trchi_component	**char ***	Current component name. Valid only when processing a component trace log file.
trchi_logfile	**char ***	Current file name.
trchi_dupcount	**int**	Number of events consumed by this **trc_read** call.

The **trcr_flags** field contains bit flags describing characteristics of the returned data. The values are:

Item	Description
TRCRF_RAW	Raw data was read, (for example) the log object was opened with the **TRC_LOGRAW** open type. Use the raw data items in the return data, (for example) those beginning with `trcri_`.
TRCRF_PROC	Processed data was read, (for example) the log object was opened with the **TRC_LOGPROC** open type. Use the processed data items in the return data, (for example) those beginning with `trchi_`.
TRCRF_64BIT	The data is from a 64-bit environment. Note that the trace itself may be from a 32 or 64 bit kernel.
TRCRF_TIMESTAMPED	The entry was timestamped when traced.
TRCRF_CPUIDOK	The cpu id is known. This is always set for a processed entry, and set for a raw entry if the cpuid was contained in each trace hook (see the **-p trace** command option), or the trace is a multi-cpu trace (see the **-C** trace option). For a processed trace, the cpu id may not be accurate if the appropriate hooks, 106 and 10C, weren't traced.
TRCRF_GENERIC	This is a generic trace entry, one traced with the **TRCGEN** or **TRCGENT** macros. This is set for a raw trace only.
TRCRF_64BITTRACE	This is a 64-bit trace, (for example) it was taken with a 64-bit kernel.
TRCRF_LIVEDATA	The data is live, don't free it. The data will be changed when another read operation is done.
TRCRF_NOPRINT	The associated trace template specified **$NOPRINT** or **$SKIP**, (for example) no data should be printed.

Return Values

Upon successful completion, the **trc_read** subroutine returns a 0 and puts the data into the *ret* area.

Error Codes

Upon error, the **trc_read** subroutine sets the **errno** global variable to a value from **errno.h**, and returns either a value from the **errno.h** file or an error defined in the **libtrace.h** file.

Item	Description
EINVAL	The handle is not valid.
TRCE_BADFORMAT	The trace data is improperly formatted, and the **errno** global variable is set to **EINVAL**.

trc_reg Subroutine

Purpose

Returns register values.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_reg(handle, regid, ret)
trc_log_handle_t handle;
int regid;
uint64_t *ret;
```

Description

The **trc_reg** subroutine is used to retrieve machine-programmable register values from either a processed or raw trace entry. It returns a -1 if the specified item was not traced.

trc_reg is only valid for a 64-bit kernel trace.

Parameters

Item	Description
handle	Contains the handle returned from a successful trc_open.
regid	One of the following reserved register identifiers found in **libtrace.h**: **TRC_PURR_ID** The PURR register. **TRC_SPURR_ID** The SPURR register. **TRC_MCR0_ID, TRC_MCR1_ID, TRC_MCRA_ID** The MCR registers, 0, 1, and A. **TRC_PMCn_ID** PMC register *n*, where *n* is a value from 1 to 8
ret	Points to an unsigned 64-bit integer to hold the return data. If the PURR is returned, it is returned in the same units as the elapsed time (that is, ticks for a raw trace and nanoseconds for a processed trace).

Return Values

The **trc_reg** subroutine returns 0 on success; otherwise, it returns the **errno** value.

Error Codes

Item	Description
EINVAL	The specified register ID is invalid.
TRCE_EOF	The specified register ID is valid but was not traced. **Note: TRCE_EOF** is the libtrace error for EOF or not found.

Related Information

The trace daemon and trcrptcommand.

trc_seek and trc_tell Subroutine

Purpose

Seeks into a trace object and returns the current position that will be used with a future seek.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

int trc_seek (handle, log_positionp, r)
trc_loghandle_t handle;
trc_logpos_t log_positionp;
trc_read_t *r;
```

```
int trc_tell (handle, log_positionp)
trc_loghandle_t handle;
trc_logpos_t *log_positionp;
```

Description

The **trc_seek** subroutine seeks into the log object identified by the *handle* parameter. The *log_positionp* parameter must have been obtained from a previous call to the **trc_tell** subroutine. If the **trc_read_t** pointer, *r*, is not NULL, the **trc_seek** subroutine returns the trace data at the seek point.

The **trc_tell** subroutine creates a **trc_logpos_t** object using the current log position and state.

The **trc_free** subroutine should be used to free a **trc_logpos_t** object that's no longer needed. However, **trc_free** is not necessary if the **trc_logpos_t** object is passed to another **trc_tell**.

Parameters

Item	Description
handle	Contains the handle returned from a successful call to the **trc_open** subroutine.
log_positionp	A **trc_logpos_t** returned by a previous call to the **trc_tell** subroutine.
r	If not NULL, points to a **trc_read_t data** item where the data at the new position is returned.

Return Values

Upon successful return, the **trc_seek** and **trc_tell** subroutines return 0.

Error Codes

If unsuccessful, the **trc_seek** subroutine returns an i/o error, or **EINVAL** if either the *handle* or *log_positionp* parameter is in error.

Upon error, the **trc_tell** subroutine returns **EINVAL** if the handle is invalid, or **ENOMEM** if storage can't be obtained for the **trc_logpos_t** object.

trc_strerror Subroutine

Purpose

Returns the error message, or next error message, associated with a trace log object or **trc_loginfo** object.

Library

libtrace.a

Syntax

```
#include <sys/libtrace.h>

char *trc_strerror (handle, rv)
void *handle;
int rv;
```

Description

The **trc_strerror** subroutine is similar to the **strerror** subroutine. If the error in the *rv* parameter is an error from the **errno.h** file, it simply returns the string from the **strerror** subroutine. If the *rv* parameter is a **libtrace** error such as **TRCE_EOF**, it returns the string associated with this error. It is possible for multiple **libtrace** errors to be present. The **trc_strerror** subroutine returns the next error in this case. When no more errors are present, the **trc_strerror** subroutine returns NULL.

Like the **strerror** subroutine, the **trc_strerror** subroutine must not be used in a threaded environment.

Parameters

Item	Description
handle	Contains the handle returned from the **trc_open** subroutine, the pointer to a **trc_loginfo_t** object, or NULL. If a handle returned by the **trc_open** subroutine is passed, the **trc_open** subroutine need not have been successful, but the **TRC_RETAIN_HANDLE** open option must have been used.
rv	Contains the return value from a call to the **libtrace** subroutine.

Return Values

The **trc_strerror** subroutine returns a pointer to the associated error message. It returns NULL if no more errors are present.

Examples

1. To retrieve all error messages from a call to the **trc_open** subroutine, call the **trc_strerror** subroutine as follows:

```
{
   trc_loghandle_t h;
   int rv;
   char *fn, *tfn, *s;

    ...

    rv = trc_open(fn,tfn, TRC_LOGREAD|TRC_LOGPROC|TRC_RETAIN_HANDLE, &h);
    while (rv && s=trc_strerror(h, rv))  {
      fprintf(stderr, "%s\n", s);
    }
}
```

2. To accomplish the same thing as the previous example with a single call, do the following:

```
{
   trc_loghandle_t h;
   int rv;
   char *fn, *tfn;

   ...

   rv = trc_open(fn,tfn, TRC_LOGREAD|TRC_LOGPROC|TRC_RETAIN_HANDLE, &h);
   if (rv) trc_perror(h, rv, "");
   }
```

trcgen or trcgent Subroutine

Purpose

Records a trace event for a generic trace channel.

Library

Runtime Services Library (**librts.a**)

Syntax

```
#include <sys/trchkid.h>
```

void trcgen(*Channel***, *HkWord*, *DataWord*, *Length*, *Buffer*)**
unsigned int *Channel*, *HkWord*, *DataWord*, *Length*;
char * *Buffer*;

```
void trcgent(Channel, HkWord, DataWord, Length, Buffer)
unsigned int Channel, HkWord, DataWord, Length;
char *Buffer;
```

Description

The **trcgen** subroutine records a trace event for a generic trace entry consisting of a hook word, a data word, a variable number of bytes of trace data and, beginning with AIX 5L Version 5.3 with the 5300-05 Technology Level, a time stamp. The **trcgent** subroutine records a trace event for a generic trace entry consisting of a hook word, a data word, a variable number of bytes of trace data, and a time stamp.

The **trcgen** subroutine and **trcgent** subroutine are located in pinned kernel memory.

Parameters

Item	Description
Buffer	Specifies a pointer to a buffer of trace data. The maximum size of the trace data is 4096 bytes.
Channel	Specifies a channel number for the trace session, obtained from the **trcstart** subroutine.
DataWord	Specifies a word of user-defined data.

Item	Description
HkWord	Specifies an integer consisting of two bytes of user-defined data (*HkData*), a hook ID (*HkID*), and a hook type (*Hk_Type*). ***HkData*** Specifies two bytes of user-defined data. ***HkID*** Specifies a hook identifier. For applications before AIX 6.1 and 32-bit applications running on AIX 6.1 and later, the hook ID value ranges from hex 010 through hex 0FF. For 64-bit applications running on AIX 6.1 and later, the hook ID value ranges from hex 0100 through hex 0FF0.
Length	Specifies the length in bytes of the *Buffer* parameter.

trchook, utrchook, trchook64, and utrhook64 Subroutine

Purpose

Records a trace event.

Library

Runtime Services Library (**librts.a**)

Syntax

```
#include <sys/trchkid.h>
```

void trchook(*HkWord*, *d1*, *d2*, *d3*, *d4*, *d5***)**
unsigned int *HkWord*, *d1*, *d2*, *d3*, *d4*, *d5*;

```
void utrchook(HkWord, d1, d2, d3, d4, d5)
unsigned int HkWord, d1, d2, d3, d4, d5;
```

```
void trchook64 (HkWord, d1, d2, d3, d4, d5)
unsigned long HkWord, d1, d2, d3, d4, d5;
```

```
void utrchook64 (HkWord, d1, d2, d3, d4, d5)
unsigned long HkWord, d1, d2, d3, d4, d5;
```

Description

The **trchook** subroutine records a trace event if a trace session is active. Input parameters include a hook word (*HkWord*) and from 0 to 5 words of data. The **trchook** and **trchook64** subroutines are intended for use by the kernel and extensions.

The **utrchook** and **utrchook64** subroutines are intended for programs running at user (application) level.

The **trchook** and **utrchook** subroutines are for use in a 32-bit environment, while the **trchook64** and **utrchook64** subroutines are intended for use in a 64-bit environment. Note that if running a 64-bit application on a 32-bit kernel, the application should use **utrchook64**(the subroutine for its 64-bit environment).

It is strongly recommended that the C macros **TRCHKLn** and **TRCHKLnT** (where **n** is from 0 to 5) be used if possible, instead of calling these subroutines directly.

Beginning with AIX 5L Version 5.3 with the 5300-05 Technology Level, all events are implicitly appended with a time stamp.

Parameters

Item	Description
d1, d2, d3, d4, d5	Up to 5 words of data from the calling program.
HkWord	The *HkWord* parameter has a different format based upon the environment. For the **trchook** and **utrchook** subroutines, it is an unsigned long consisting of a hook ID (*HkID*), a hook type (*Hk_Type*), and two bytes of data from the calling program (*HkData*). ***HkID*** A hook ID is a 12-bit value. For user programs, the hook ID may be a value from 0x010 to 0x0FF. Hook identifiers are defined in the **/usr/include/sys/trchkid.h** file. ***Hk_Type*** A 4-bit value that identifies the amount of trace data to be recorded: **Value** — Records **1** — Hook word **9** — Hook word and a time stamp **2** — Hook word and one data word **A** — Hook word, one data word, and a time stamp **6** — Hook word and up to five data words **E** — Hook word, up to five data words, and a time stamp. ***HkData*** Two bytes of data from the calling program.

In a 64-bit environment, when the **trchook64** or **utrchook64** subroutine is used, the format is *ffffllllhhhhssss*, where *f* represents flags, *l* is length, *h* is the hook ID, and *s* is the subhook.

Beginning with AIX 6.1, 16-bit hook IDs are available in the 64-bit environment. 16-bit hook IDs in the form of 0x*hhh*0 are equivalent to 12-bit hook IDs in the form of 0x*hhh* where *h* is a hexadecimal digit. When a hook ID is less than 0x1000, its least significant digit must be 0.

The hook and subhook ids are the same as for the 32-bit environment (12-bit hook id and a 16-bit subhook id). Note that the 4 bits between the hook id and subhook are unused.

The flags (the first 16 bits of the 64-bit hookword) are specified as follows:

8000
The hook should be timestamped.

4000
A generic trace entry, should not use the **trchook64** or **utrchook64** subroutine. For more information see “trcgen or trcgent Subroutine” on page 2226.

2000
The hook contains 32-bit data. Used by aix trace only.

1000
Automatically include the cpuid when tracing the data.

The length (*l*) is the second 16 bits of the hookword. It is the length of the data. The length is 0 if no data other than the hookword is traced (**TRCHKL0**), 8 if one parameter, 8 bytes, is traced (**TRCHKL1**), 16 for 2 parameters, 24 for 3 parameters, 32 for 4 parameters, and 40 for 5 parameters (**TRCHKL5**).

trcoff Subroutine

Purpose

Halts the collection of trace data from within a process.

Library

Runtime Services Library (**librts.a**)

Syntax

```
int trcoff( Channel)
int Channel;
```

Description

The **trcoff** subroutine stops trace data collection for a trace channel. The trace session must have already been started using the **trace** command or the **trcstart** subroutine.

Parameters

Item	Description
Channel	Channel number for the trace session.

Return Values

If the **trcoff** subroutine was successful, zero is returned and trace data collection stops. If unsuccessful, a negative one is returned.

trcon Subroutine

Purpose

Starts the collection of trace data.

Library

Runtime Services Library (**librts.a**)

Syntax

```
int trcon( Channel)
int Channel;
```

Description

The **trcon** subroutine starts trace data collection for a trace channel. The trace session must have already been started using the **trace** command or the **trcstart** (“trcstart Subroutine” on page 2230) subroutine.

Parameters

Item	Description
Channel	Specifies one of eight trace channels. Channel number 0 always refers to the Event/ Performance trace. Channel numbers 1 through 7 specify generic trace channels.

Return Values

If the **trcon** subroutine was successful, zero is returned and trace data collection starts. If unsuccessful, a negative one is returned.

trcstart Subroutine

Purpose

Starts a trace session.

Library

Runtime Services Library (**librts.a**)

Syntax

```
int trcstart( Argument)
char *Argument;
```

Description

The **trcstart** subroutine starts a trace session. The *Argument* parameter points to a character string containing the flags invoked with the **trace** daemon. To specify that a generic trace session is to be started, include the **-g** flag.

Parameters

Item	Description
Argument	Character pointer to a string holding valid arguments from the **trace** daemon.

Return Values

If the **trace** daemon is started successfully, the channel number is returned. Channel number 0 is returned if a generic trace was not requested. If the **trace** daemon is not started successfully, a value of -1 is returned.

When there is no privilege to run channel 0, the **trcstart** subroutine returns a value of -1.

Files

Item	Description
/dev/trace	Trace special file.

trcstop Subroutine

Purpose

Stops a trace session.

Library

Runtime Services Library (**librts.a**)

Syntax

```
# include <sys/trcmacros.h>
# define TRCSTOP SERIAL 0x40000000
# define TRCSTOP DISCARDBUFS 0x20000000
int trcstop( Channel)
int Channel;
```

Description

The **trcstop** subroutine stops a trace session for a particular trace channel.

Parameters

Item	Description
Channel	Specifies one of eight trace channels. Channel number 0 always refers to the Event/Performance trace. Channel numbers 1 through 7 specify generic trace channels.
Serial (TRCSTOP SERIAL)	If the channel is ORed with the *Serial* flag, then the **trcstop** subroutine serializes the trace I/O operations from multiple processor buffers into the trace file. The *Serial* flag is applicable for all modes of tracing. This flag is mutually exclusive with the *discard_buff* flag.
discard_buff (TRCSTOP DISCARDBUFF0)	To set this option, the user needs to OR the *discard_buff* flag with the channel option. When invoked, the **trcstop** subroutine discards any captured trace buffers pending I/O operation. If trace buffers have already been written into file, then the *discard_buff* flag is ignored. This flag is mutually exclusive with the serial flag.

Return Values

Item	Description
0	The trace session was stopped successfully.
-1	The trace session did not stop.

trunc, truncf, truncl, truncd32, truncd64, or truncd128 Subroutine

Purpose

Rounds to truncated integer value.

Syntax

```
#include <math.h>

double trunc (x)
double x;
float truncf (x)
float x;

long double truncl (x)
long double x;

_Decimal32 truncd32(x)
_Decimal32 x;

_Decimal64 truncd64(x)
_Decimal64 x;

_Decimal128 truncd128(x)
_Decimal128 x;
```

Description

The **trunc**, **truncf**, **truncl**, **truncd32**, **truncd64**, and **truncd128** subroutines round the *x* parameter to the integer value, in floating format, nearest to but no larger in magnitude than the *x* parameter.

Parameters

Item	Description
x	Specifies the value to be rounded.

Return Values

Upon successful completion, the **trunc**, **truncf**, **truncl**, **truncd32**, **truncd64**, and **truncd128** subroutines return the truncated integer value.

If *x* is NaN, a NaN is returned.

If *x* is ±0 or ±Inf, *x* is returned.

truncate, truncate64, ftruncate, or ftruncate64 Subroutine

Purpose

Changes the length of regular files or shared memory object.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
int truncate ( Path,  Length)
const   char *Path;
off_t Length;
```

```
int ftruncate ( FileDescriptor, Length)
int FileDescriptor;
off_t Length;
```

```
int truncate64 ( Path,  Length)
const   char *Path;
off64_t Length;

int ftruncate64 ( FileDescriptor, Length)
int FileDescriptor;
off64_t Length;
```

Description

The **truncate** and **ftruncate** subroutines change the length of regular files or shared memory object.

The *Path* parameter must point to a regular file for which the calling process has write permission. The *Length* parameter specifies the wanted length of the new file in bytes.

The *Length* parameter measures the specified file in bytes from the beginning of the file. If the new length is less than the previous length, all data between the new length and the previous end of file is removed. If the new length in the specified file is greater than the previous length, data between the old and new lengths is read as zeros. Full blocks are returned to the file system so that they can be used again, and the file size is changed to the value of the *Length* parameter.

If the file designated in the *Path* parameter names a symbolic link, the link is traversed and path name resolution continues.

These subroutines do not modify the seek pointer of the file.

These subroutines cannot be applied to a file that a process has open with the **O_DEFER** flag.

Successful completion of the **truncate** or **ftruncate** subroutine updates the `st_ctime` and `st_mtime` fields of the file. Successful completion also clears the SetUserID bit (**S_ISUID**) of the file if any of the following are true:

- The calling process does not have root user authority.
- The effective user ID of the calling process does not match the user ID of the file.
- The file is executable by the group (**S_IXGRP**) or others (**S_IXOTH**).

These subroutines also clear the **SetGroupID** bit (**S_ISGID**) if the following conditions are true:

- The file does not match the effective group ID or one of the supplementary group IDs of the process
- OR
- The file is executable by the owner (**S_IXUSR**) or others (**S_IXOTH**).

  Note: Clearing of the **SetUserID** and **SetGroupID** bits can occur even if the subroutine fails because the data in the file was modified before the error was detected.

truncate and **ftruncate** can be used to specify any size up to **OFF_MAX**. **truncate64** and **ftruncate64** can be used to specify any length up to the maximum file size for the file.

In the large file enabled programming environment, **truncate** is redefined to be **truncate64** and **ftruncate** is redefined to be **ftruncate64**.

Parameters

Item	Description
Path	Specifies the name of a file that is opened, truncated, and then closed.
FileDescriptor	Specifies the descriptor of a file or shared memory object that must be open for writing.
Length	Specifies the new length of the truncated file in bytes.

Return Values

Upon successful completion, a value of 0 is returned. If the **truncate** or **ftruncate** subroutine is unsuccessful, a value of *-1* is returned and the **errno** global variable is set to indicate the nature of the error.

Error Codes

The **truncate** and **ftruncate** subroutines fail if the following is true:

Item	Description
EROFS	An attempt was made to truncate a file that occupies a read-only file system.

Note: In addition, the **truncate** subroutine can return the same errors as the **open** subroutine if a problem occurs while the file is being opened.

The **truncate** and **ftruncate** subroutines fail if one of the following is true:

Item	Description
EAGAIN	The truncation operation fails when an enforced write lock on a portion of the file that is being truncated. Because the target file was opened with the **O_NONBLOCK** or **O_NDELAY** flags set, the subroutine fails immediately rather than wait for a release.
EDQUOT	New disk blocks cannot be allocated for the truncated file. The quota of the user's or group's allotted disk blocks was exhausted on the target file system.
EFBIG	An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. If the user environment variable **XPG_SUS_ENV=ON** is set before execution of the process, then the **SIGXFSZ** signal is posted to the process when it exceeds the process' file size limit.
EFBIG	The file is a regular file and *length* is greater than the offset maximum established in the open file description that is associated with *fildes*.
EINVAL	The file is not a regular file.
EINVAL	The *Length* parameter was less than zero.
EISDIR	The named file is a directory.
EINTR	A signal was caught during execution.
EIO	An I/O error occurred while reading from or writing to the file system.
EMFILE	The file is open with **O_DEFER** by one or more processes.
ENOSPC	New disk blocks cannot be allocated for the truncated file. There is no free space on the file system that contains the file.
ETXTBSY	The file is part of a process that is running.
EROFS	The named file occupies a read-only file system.

Note:

1. The **truncate** subroutine can also be unsuccessful for other reasons. For a list of more errors, see Base Operating System error codes for services that require path-name resolution.
2. The **truncate** subroutine can return the same errors as the **open** subroutine if a problem occurs while the file is being opened.

The **ftruncate** subroutine fails if the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter is not a valid file descriptor open for writing.

Item	Description
EINVAL	The *FileDescriptor* argument references a file that was opened without write permission.

The **truncate** function fails if the following conditions are true:

Item	Description
EACCES	A component of the path prefix denies search permission, or write permission is denied on the file.
EISDIR	The named file is a directory.
ELOOP	Too many symbolic links were encountered in resolving *path*.
ENAMETOOLONG	The length of the specified path name exceeds **PATH_MAX** bytes, or the length of a component of the path name exceeds **NAME_MAX** bytes.
ENOENT	A component of *path* does not name an existing file or *path* is an empty string.
ENTDIR	A component of the path prefix of *path* is not a directory.
EROFS	The named file occupies a read-only file system.

The **truncate** function fails if the following is true:

Item	Description
ENAMETOOLONG	Path name resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.

tsearch, tdelete, tfind or twalk Subroutine

Purpose

Manages binary search trees.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <search.h>
```

```
void *tsearch ( Key,  RootPointer,  ComparisonPointer)
const void *Key;
void **RootPointer;
int (*ComparisonPointer) (const void *Element1, const void *Element2);
```

```
void *tdelete (Key, RootPointer, ComparisonPointer)
const void *Key;
void **RootPointer;
int (*ComparisonPointer) (const void *Element1, const void *Element2);
```

```
void *tfind (Key, RootPointer, ComparisonPointer)
const void *Key;
void *const *RootPointer;
int (*ComparisonPointer) (const void *Element1, const void *Element2);
```

```
void twalk ( Root,  Action)
const void *Root;
void (*Action) (const void *Node, VISIT Type, int Level);
```

Description

The **tsearch**, **tdelete**, **tfind** and **twalk** subroutines manipulate binary search trees. Comparisons are made with the user-supplied routine specified by the *ComparisonPointer* parameter. This routine is called with two parameters, the pointers to the elements being compared.

The **tsearch** subroutine performs a binary tree search, returning a pointer into a tree indicating where the data specified by the *Key* parameter can be found. If the data specified by the *Key* parameter is not found, the data is added to the tree in the correct place. If there is not enough space available to create a new node, a null pointer is returned. Only pointers are copied, so the calling routine must store the data. The *RootPointer* parameter points to a variable that points to the root of the tree. If the *RootPointer* parameter is the null value, the variable is set to point to the root of a new tree. If the *RootPointer* parameter is the null value on entry, then a null pointer is returned.

The **tdelete** subroutine deletes the data specified by the *Key* parameter. The *RootPointer* and *ComparisonPointer* parameters perform the same function as they do for the **tsearch** subroutine. The variable pointed to by the *RootPointer* parameter is changed if the deleted node is the root of the binary tree. The **tdelete** subroutine returns a pointer to the parent node of the deleted node. If the data is not found, a null pointer is returned. If the *RootPointer* parameter is null on entry, then a null pointer is returned.

The **tfind** subroutine searches the binary search tree. Like the **tsearch** subroutine, the **tfind** subroutine searches for a node in the tree, returning a pointer to it if found. However, if it is not found, the **tfind** subroutine will return a null pointer. The parameters for the **tfind** subroutine are the same as for the **tsearch** subroutine.

The **twalk** subroutine steps through the binary search tree whose root is pointed to by the *RootPointer* parameter. (Any node in a tree can be used as the root to step through the tree below that node.) The *Action* parameter is the name of a routine to be invoked at each node. The routine specified by the *Action* parameter is called with three parameters. The first parameter is the address of the node currently being pointed to. The second parameter is a value from an enumeration data type:

```
typedef enum [preorder, postorder, endorder, leaf] VISIT;
```

(This data type is defined in the **search.h** file.) The actual value of the second parameter depends on whether this is the first, second, or third time that the node has been visited during a depth-first, left-to-right traversal of the tree, or whether the node is a *leaf*. A leaf is a node that is not the parent of another node. The third parameter is the level of the node in the tree, with the root node being level zero.

Although declared as type pointer-to-void, the pointers to the key and the root of the tree should be of type pointer-to-element and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Parameters

Item	Description
Key	Points to the data to be located.
ComparisonPointer	Points to the comparison function, which is called with two parameters that point to the elements being compared.
RootPointer	Points to a variable that in turn points to the root of the tree.
Action	Names a routine to be invoked at each node.
Root	Points to the roots of a binary search node.

Return Values

The comparison function compares its parameters and returns a value as follows:

- If the first parameter is less than the second parameter, the *ComparisonPointer* parameter returns a value less than 0.
- If the first parameter is equal to the second parameter, the *ComparisonPointer* parameter returns a value of 0.
- If the first parameter is greater than the second parameter, the *ComparisonPointer* parameter returns a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

If the node is found, the **tsearch** and **tfind** subroutines return a pointer to it. If the node is not found, the **tsearch** subroutine returns a pointer to the inserted item and the **tfind** subroutine returns a null pointer. If there is not enough space to create a new node, the **tsearch** subroutine returns a null pointer.

If the *RootPointer* parameter is a null pointer on entry, a null pointer is returned by the **tsearch** and **tdelete** subroutines.

The **tdelete** subroutine returns a pointer to the parent of the deleted node. If the node is not found, a null pointer is returned.

tss_create Subroutine

Purpose

This subroutine creates a thread-specific storage pointer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
int tss_create(tss_t *key, tss_dtor_t dtor);
```

Description

The **tss_create** subroutine creates a thread-specific storage pointer with the **dtor** destructor, which is potentially null.

Parameters

Item	Description
key	A thread-specific storage pointer that is created.
dtor	A pointer for a destructor and it is potentially null.

Return Values

If the **tss_create** subroutine is successful, it sets the value of the **key** thread-specific storage pointer that uniquely identifies the newly created pointer and returns **thrd_success**. If the **tss_create** subroutine fails the **thrd_error** is returned and the value of the **key** thread-specific storage pointer is set to an undefined value.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

tss_delete Subroutine

Purpose

This subroutine deletes a thread-specific storage pointer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
void tss_delete(tss_t key);
```

Description

The **tss_delete** subroutine releases any resources that are used by the thread-specific storage pointer that is identified by the **key** parameter.

Parameters

Item	Description
key	Holds an identifier for a thread-specific storage pointer.

Return Values

The **tss_delete** subroutine returns no value.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

tss_get Subroutine

Purpose

This subroutine fetches the thread-specific storage pointer that is based on the **key** value.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
void *tss_get(tss_t key);
```

Description

The **tss_get** function returns the value for the current thread that is held in the thread-specific storage pointer that is identified by the **key** parameter.

Parameters

Item	Description
key	Holds a thread-specific storage pointer.

Return Values

The **tss_get** function returns the value for the current thread if successful or it returns zero if the **tss_get** function is unsuccessful.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

tss_set Subroutine

Purpose

This subroutine sets the value of the **val** parameter in the thread-specific storage pointer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <threads.h>
```

```
int tss_set(tss_t key, void *val);
```

Description

The **tss_set** function sets the value for the current thread that is held in the thread-specific storage pointer that is identified by the **key** parameter to the **val** parameter.

Parameters

Item	Description
key	Holds a thread-specific storage pointer.
key	Holds the value to be set in the thread-specific storage pointer.

Return Values

The **tss_set** function returns **thrd_success** on success or it returns **thrd_error** if the request is not completed.

Files

Item	Description
threads.h	Standard macros, data types, and subroutines are defined by the **threads.h** file.

ttylock, ttywait, ttyunlock, or ttylocked Subroutine

Purpose

Controls tty locking functions.

Library

Standard C Library (**libc.a**)

Syntax

```
int ttylock ( DeviceName)
char *DeviceName;
```

```
int ttywait (DeviceName)
char *DeviceName;
```

```
int ttyunlock (DeviceName)
char *DeviceName;
```

```
int ttylocked (DeviceName)
char *DeviceName;
```

Description

The **ttylock** subroutine creates the **LCK..***DeviceName* file in the **/etc/locks** directory and writes the process ID of the calling process in that file. If **LCK..***DeviceName* exists and the process whose ID is contained in this file is active, the **ttylock** subroutine returns an error.

There are programs like **uucp** and **connect** that create tty locks in the **/etc/locks** directory. The convention followed by these programs is to call the **ttylock** subroutine with an argument of *DeviceName* for locking the **/dev/***DeviceName* file. This convention must be followed by all callers of the **ttylock** subroutine to make the locking mechanism work.

The **ttywait** subroutine blocks the calling process until the lock file associated with *DeviceName*, the **/etc/locks/LCK..***DeviceName* file, is removed.

The **ttyunlock** subroutine removes the lock file, **/etc/locks/LCK..***DeviceName*, if it is held by the current process.

The **ttylocked** subroutine checks to see if the lock file, **/etc/locks/LCK..***DeviceName*, exists and the process that created the lock file is still active. If the process is no longer active, the lock file is removed.

Parameters

Item	Description
DeviceName	Specifies the name of the device.

Return Values

Upon successful completion, the **ttylock** subroutine returns a value of 0. Otherwise, a value of -1 is returned.

The **ttylocked** subroutine returns a value of 0 if no process has a lock on device. Otherwise, a value of -1 is returned.

Examples

1. To create a lock for **/dev/tty0**, use the following statement:

```
rc = ttylock("tty0");
```

2. To lock **/dev/tty0** device and wait for lock to be cleared if it exists, use the following statements:

```
if (ttylock("tty0"))
    ttywait("tty0");
rc = ttylock("tty0");
```

3. To remove the lock file for device **/dev/tty0** created by a previous call to the **ttylock** subroutine, use the following statement:

```
ttyunlock("tty0");
```

4. To check for a lock on **/dev/tty0**, use the following statement:

```
rc = ttylocked("tty0");
```

ttyname or isatty Subroutine

Purpose

Gets the name of a terminal or determines if the device is a terminal.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>
```

```
char *ttyname( FileDescriptor)
int FileDescriptor;
```

```
int isatty(FileDescriptor)
int FileDescriptor;
```

Description

Attention: Do not use the **ttyname** subroutine in a multithreaded environment.

The **ttyname** subroutine gets the path name of a terminal.

The **isatty** subroutine determines if the file descriptor specified by the *FileDescriptor* parameter is associated with a terminal.

The **isatty** subroutine does not necessarily indicate that a person is available for interaction, since nonterminal devices may be connected to the communications line.

Parameters

Item	Description
FileDescriptor	Specifies an open file descriptor.

Return Values

The **ttyname** subroutine returns a pointer to a string containing the null-terminated path name of the terminal device associated with the file descriptor specified by the *FileDescriptor* parameter. A null pointer is returned and the **errno** global variable is set to indicate the error if the file descriptor does not describe a terminal device in the **/dev** directory.

The return value of the **ttyname** subroutine may point to static data whose content is overwritten by each call.

If the specified file descriptor is associated with a terminal, the **isatty** subroutine returns a value of 1. If the file descriptor is not associated with a terminal, a value of 0 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **ttyname** and **isatty** subroutines are unsuccessful if one of the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.
ENOTTY	The *FileDescriptor* parameter does not specify a terminal device.

Files

Item	Description
/dev/*	Terminal device special files.

ttyslot Subroutine

Purpose

Finds the slot in the **utmp** file for the current user.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
int ttyslot (void)
```

Description

The **ttyslot** subroutine returns the index of the current user's entry in the **/etc/utmp** file. The **ttyslot** subroutine scans the **/etc/utmp** file for the name of the terminal associated with the standard input, the standard output, or the error output file descriptors (0, 1, or 2).

The **ttyslot** subroutine returns -1 if an error is encountered while searching for the terminal name, or if none of the first three file descriptors (0, 1, and 2) is associated with a terminal device.

Files

Item	Description
/etc/inittab	The path to the **inittab** file, which controls the initialization process.
/etc/utmp	The path to the **utmp** file, which contains a record of users logged in to the system.

typeahead Subroutine

Purpose

Controls checking for typeahead.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

int typeahead
(int fildes);
```

Description

The **typeahead** subroutine controls the detection of typeahead during a refresh, based on the value of *fildes*:

- If *fildes* is a valid file descriptor, the **typeahead** subroutine is enabled during refresh; Curses periodically checks *fildes* for input and aborts refresh if any character is available. (This is the initial setting, and the *typeahead* file descriptor corresponds to the input file associated with the screen created by the **initscr** or **newterm** subroutine.) The value of *fildes* need not be the file descriptor on which the refresh is occurring.
- If *fildes* is -1, Curses does not check for typeahead during refresh.

Parameters

Item	Description
fildes	

Return Value

Upon successful completion, the **typeahead** subroutine returns OK. Otherwise, it returns ERR.

Example

To turn typeahead checking on, enter:

```
typeahead(1);
```

u

The following Base Operating System (BOS) runtime services begin with the letter *u*.

ukey_enable Subroutine

Purpose

Enables user-keys in a process.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/ukeys.h>

int ukey_enable (void)
```

Description

The **ukey_enable** subroutine allows a process access to the user-keys memory protection facilities. A process must make a successful call to the **ukey_enable** subroutine to enable user-keys before attempting other user-key specific APIs. The following are necessary conditions for enabling user-keys for a process:

1. Running with the 64-bit kernel. User-keys are not supported on the 32-bit kernel.
2. Running on hardware that supports storage-keys and user-keys have not been explicitly disabled. By default, user-keys are enabled if the platform supports it. The **sysconf** (*_SC_AIX_UKEYS*) subroutine returns the number of available user-keys.
3. If multi-threaded, the process must be running in system-scope such as 1:1 mode.
4. Process is not checkpointable for Load Leveler dispatched jobs.

All threads of a user-key enabled process are initially set-up with an active user-key-set that only allows both read and write access to memory pages that have been assigned to the **UKEY_PUBLIC**, the default user-key. Individual threads can modify their active user-key-set by calling user-key APIs to construct and activate user-key-sets.

Signal Context for User-Key-Enabled processes:

The default signal context for a user-key-enabled process is modified for any future signals that are received. The ***ucontext_t*** structure is extended to include the active user-key-set of the interrupted thread. This is provided to signal handlers.

Note: Although signal handlers take a pointer to the ***sigcontext_t*** as per the documentation for the**sigaction** subroutine, the actual structure constructed on the stack is the ***ucontext_t*** structure, which is a superset of the ***sigcontext_t*** and matches it in its initial portion. By pointing at the signal context with an ***ucontext_t*** pointer, signal handlers might access the extended data.

The following fields are set:

```
ucontext_t.__extctx.__flags |= __EXTCTX_UKEYS
ucontext_t.__extctx.__ukeys[2] = active user-key-set
```

The user-key extended context is independent of VMX context and is built for all processes that are user-key-enabled.

Additionally, if a storage key exception is taken, the exception type field is set to indicate this in the extended context:

```
ucontext_t.uc_mcontext.jmp_context.excp_type = EXCEPT_SKEY.
```

See the *sys/context.h* header file for a more detailed layout of the extended context information.

Return Values

When successful, the **ukey_enable** subroutine returns the number of available user-keys. Otherwise, it returns a value of -1 and sets the **errno** global variable to indicate the error.

Errors Codes

Item	Description
ENOSYS	User-keys are not supported.

Related Information

The **ukey_setjmp** subroutine.

The **ukeyset_init** subroutine.

The **ukeyset_add_key, ukeyset_remove_key, ukeyset_add_set, ukeyset_remove_set** subroutine.

The **ukeyset_activate** subroutine.

The **ukeyset_ismember** subroutine.

The **pthread_attr_getukeyset_np or pthread_attr_setukeyset_np** subroutine.

AIX Vector Programming in*General Programming Concepts: Writing and Debugging Programs* .

ukeyset_add_key, ukeyset_remove_key, ukeyset_add_set or ukeyset_remove_set Subroutine

Purpose

Operates on and modifies a user-key-set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/ukeys.h>

int ukeyset_add_key (uset, key, flags)
ukeyset_t * uset;
ukey_t key;
unsigned int flags;

int ukeyset_remove_key (uset, key, flags)
ukeyset_t * uset;
ukey_t key;
unsigned int flags;

int ukeyset_add_set (uset, aset)
```

```
ukeyset_t * uset;
ukeyset_t aset;

int ukeyset_remove_set (uset, rset)
ukeyset_t * uset;
ukeyset_t rset;
```

Description

These subroutines operate on and modify user-key-sets. The user-key-set must have been originally initialized with the **ukeyset_init** subroutine.

Individual or groups (sets) of user-keys can be added or removed . When adding or removing an individual key, the accesses (read or write, or both read and write) being added or removed must be specified through the *flags* parameter. When adding or removing user-key-sets, specification is not required, because a key-set contains not only information on what keys are enabled, but also information on which specific access permissions are enabled for each one of those keys.

The **ukeyset_add_key** subroutine adds the user-key specified by the *key* parameter with accesses as specified by the *flags* parameter to the user-key-set specified by the *uset* parameter. The **ukeyset_remove_key** subroutine removes the accesses specified by the *flags* parameter of the key specified by the *key* parameter from the user-key-set specified by the *uset* parameter. The **ukeyset_add_set** subroutine adds the keys and accesses specified by the *aset* key-set parameter to the user-key-set specified by the *uset* parameter. The **ukeyset_remove_set** subroutine removes the keys and accesses specified by the *rset* key-set parameter from the user-key-set specified by the *uset* parameter.

Note: An add operation of a key (or key-set) and then a subsequent remove operation of the same key (or key-set) might not result in the original key-set. For example, if a key already exists in a key-set, adding the same key has no effect on the key-set, but then a subsequent remove key operation results in a new key-set minus the removed key.

Attempting to remove a defined user-key that does not exist in the source key-set is ignored silently in a manner similar to the signal set services.

These subroutines will fail unless the **ukey_enable** subroutine has already been successfully executed by a thread in the process. Refer to the Storage Protect Keys article for more details.

Parameters

Item	Description
uset	User-key-set to be modified.
rset	User-key-set to remove.
aset	User-key-set to add.
key	User-key to add or remove from a key-set. This parameter is combined with read or write flags when performing add or remove operations.
flags	The following flags are defined for the **ukeyset_add_key()** and **ukeyset_remove_key()** services: • **UK_READ** - Specifies that the read access for a key is to be added or removed. • **UK_WRITE** - Specifies that the write access for a key is to be added or removed. • **UK_RW** - Specifies that read and write access are to be added or removed.

Return Values

If successful, the user-key-set subroutines return a value of 0. Otherwise, they return a value of -1 and set the **errno** global variable to indicate the error.

Errors Codes

The **ukeyset_add_key** and **ukeyset_remove_key** subroutines are unsuccessful if the following are true:

Item	Description
EINVAL	Invalid *flags* parameter, invalid key-set specified in *uset* parameter or invalid (undefined) keys specified in the *key* parameter.
ENOSYS	Unconfigured (unavailable) private key specified in the *key* parameter or process is not user-key enabled.

The **ukeyset_add_set**, **ukeyset_remove_set**, `ukeyset_add_set` and `ukeyset_remove_set` subroutines are unsuccessful if the following are true:

Item	Description
EINVAL	Invalid key-set specified in *uset* , *rset* or *aset* parameter.
ENOSYS	Process is not user-key enabled.

Only the subroutines that take keys (instead of keysets) to add or remove can fail because of invalid or unused key number or invalid access flags.

ukeyset_activate Subroutine

Purpose

Activates a user-key-set and returns the previously active user-key-set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/ukeys.h>

ukeyset_t ukeyset_activate (set, command)
ukeyset_t set;
int command;
```

Description

The **ukeyset_activate** subroutine changes the currently active user-key-set and returns the previously active user-key-set. The **UKEY_PUBLIC** is always enabled for both read and write.

In POWER6 systems, the **ukeyset_activate** subroutine is implemented through a special linkage. The linkage also executes a fast-path system call. A consequence of running a fast-path system call is that the **errno** global variable is not updated for errors. Instead, the subroutine ignores some errors. For example, attempts to remove or add the **UKEY_PUBLIC** value are ignored, and if it is not ignored, the subroutine returns the **UKSET_INVALID** value.

In POWER7® systems, the **ukeyset_activate** subroutine is handled through a low memory millicode as the Authority Mask Register (AMR) is accessible in the user mode. There is no change in the way the **errno** global variable and errors are handled.

Attention: Calling this subroutine in a system that does not support storage keys or has user keys disabled results in a SIGILL signal.

Parameters

Item	Description
set	User-key-set.
command	One of the following: • **UKA_REPLACE_KEYS** - Replaces key-set with the specified key-set. • **UKA_ADD_KEYS** - Adds the specified key-set to the current key-set. • **UKA_REMOVE_KEYS** - Removes the specified key-set from the active key-set. • **UKA_GET_KEYS** - Reads the current key-set value without updating the current key-set. The input key-set is ignored.

Return Values

Upon success, the **ukeyset_activate** subroutine returns the previously active user-key-set. If called with the **UKA_GET_KEYS** command, this will also be the current active key-set. If unsuccessful, the **ukeyset_activate** key-set returns a value of the **UKSET_INVALID**.

Errors Codes

The **ukeyset_activate** subroutine does not update **errno** if unsuccessful.

Related Information

The **ukey_enable** subroutine.

ukey_setjmp Subroutine

Purpose

Saves the current execution context and active user-key-set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <setjmp.h>
#include <sys/ukeys.h>

int ukey_setjmp (ukey_context)
ukey_jmp_buf ukey_context;
```

Description

The **ukey_setjmp** subroutine saves the current stack context and signal mask and additionally saves the current active user-key-set in the *ukey_context* special jump buffer.

The *ukey_context* can be passed as a parameter to the **longjmp** subroutine, which restores not only the execution context but also the saved user-key-set.

Parameters

Item	Description
ukey_context	Specifies the address for a *ukey_jmp_buf* structure.

Return Values

The **ukey_setjmp** subroutine returns a value of 0, unless the return is from a call to the **longjmp** function, in which case the **ukey_setjmp** subroutine returns a nonzero value.

ukeyset_init Subroutine

Purpose

Initializes a user-key-set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/ukeys.h>

int ukeyset_init (nset, flags)
ukeyset_t * nset;
unsigned int flags;
```

Description

The **ukeyset_init** subroutine initializes the user-key set pointed to by the *nset* parameter. The key-set has read and write accesss enabled for **UKEY_PUBLIC** alone and disabled for all other keys. If the **UK_INIT_ADD_PRIVATE** flag is specified, read and write access for all available private user-keys is enabled.

Parameters

Item	Description
nset	Points to the user-key-set to be initialized.
flags	Must be set to zero for default behavior (only public user-key enabled) or to **UK_INIT_ADD_PRIVATE** if all private user-keys are also to be enabled.

Return Values

If successful, the **ukeyset_init** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the **errno** global variable to indicate the error.

Errors Codes

The **ukeyset_init** subroutine fails if the following are true:

Item	Description
EINVAL	Not valid flags parameter, or NULL or misaligned *nset* parameter.
ENOSYS	Not a user-key enabled process.

Related Information

The **ukey_enable** subroutine.

The **ukey_setjmp** subroutine.

The **ukeyset_add_key, ukeyset_remove_key, ukeyset_add_set, ukeyset_remove_set** subroutine.

The **ukeyset_activate** subroutine.

The **ukeyset_ismember** subroutine.

The **pthread_attr_getukeyset_np or pthread_attr_setukeyset_np** subroutine.

ukeyset_ismember Subroutine

Purpose

Tests whether a key exists in a user-key-set.

Library

Standard C library (**libc.a**)

Syntax

```
#include <sys/ukeys.h>

int ukeyset_ismember (uset, ukey, flags)
ukeyset_t * uset;
ukey_t ukey;
unsigned int flags;
```

Description

The **ukeyset_ismember** subroutine tests whether the read or write access specified by the *flags* parameter for a user-key specified by the *ukey* parameter is included in the user-key-set pointed to by the *uset* parameter.

Parameters

Item	Description
uset	Points to the user-key-set.
ukey	User-key whose membership in key-set is to be tested.

Item	Description
flags	Must be set to one of the following values: • **UK_READ** - Tests for read access • **UK_WRITE** - Tests for write access • **UK_RW** - Tests for both read and write access

Return Values

Upon successful completion, the **ukeyset_ismember** subroutine returns a value of 1, if the user-key *ukey* with the specified access *flags* is present in the indicated key-set *uset*. Otherwise, it returns a value of 0. If unsuccessful, the subroutine returns a value of -1, and the **errno** global variable is set to indicate the error.

Errors Codes

The **ukeyset_ismember** subroutine fails if the following is true:

Item	Description
EINVAL	Invalid *flags* parameter or invalid *ukey* parameter or invalid key-set parameter.
ENOSYS	The process is not a user-key-enabled process.

ukey_getkey Subroutine

Purpose

Queries the user-key for application memory.

Syntax

```
#include <sys/ukeys.h>

int ukey_getkey (void * addr, ukey_t * ukey)
```

Description

The **ukey_getkey** subroutine can be used to determine the user-key associated with a memory address. The user-key returned by this service normally corresponds to the values set by the **ukey_protect** subroutine.

When an application memory can not have its user-key altered or is not part of the application address space. A user-key value of the **UKEY_SYSTEM** is returned for this memory.

Parameters

Item	Description
addr	Specifies the address of the region to be queried.
ukey	Value for the user-mode storage-key is returned here.

Return Values

When successful, the **ukey_getkey** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the `errno` global variable to indicate the error.

Error Codes

If the **ukey_getkey** subroutine is unsuccessful, the `errno` global variable might be set to one of the following values:

Item	Description
EFAULT	A part of the buffer pointed to by the *ukey* parameter is out of range or otherwise inaccessible.
ENOMEM	Address is not valid for the address space of the process.
ENOSYS	User-keys are not supported.

ukey_protect Subroutine

Purpose

Modifies memory's user-key protection.

Syntax

```
#include <sys/ukeys.h>

int ukey_protect (void * addr, size_t len, ukey_t ukey)
```

Description

Setting user-keys is available with the **ukey_protect(addr,len,prot)** protect settings. Attempts to set user-keys with the **ukey_protect** subroutine fail if keys are not implemented, or the specified user-key is not available. The **sysconf(_SC_AIX_UKEYS)** must be used to test for the number and presence of user-keys.

One user-key can be associated with a virtual page. The supported values are the **UKEY_PUBLIC** and **UKEY_PRIVATE1-31** values. A successful call to the **ukey_protect()** subroutine replaces the region's previous user-key(s) with the value specified by *ukey*.

A user-key can be set on shared memory regions (**shmat**()) and applications default data and stack region. When using the **ukey_protect** subroutine on shared memory regions, the region must have write access to the shared memory object (process with the appropriate privileges an effective user ID that matches shm_perm.uid or shm_perm.cuid). User-keys cannot be altered on files mapped with **shmat**(), application text, or library regions.

When using the `ukey_protect` subroutine to place a private key on memory that is acquired by the `malloc` subroutine, the memory must always be reset to the **UKEY_PUBLIC** key before it is freed.

Parameters

Item	Description
addr	Specifies the address of the region to be modified. Must be a multiple of the page size returned by the `vmgetinfo` subroutine using the `VM_PAGE_INFO` command.
len	Specifies the length, in bytes, of the region to be modified. If the *len* parameter is not a multiple of the page size returned by the `sysconf` subroutine using the *_SC_PAGE_SIZE* value for the *Name* parameter, an error is returned.
ukey	Specifies the user-key value to associate with the address range.

Return Values

When successful, the **ukey_protect** subroutine returns a value of 0. Otherwise, it returns a value of -1 and sets the `errno` global variable to indicate the error.

Error Codes

Attention: If the **ukey_protect** subroutine is unsuccessful because of a condition other than that specified by the **EINVAL** error code, the access protection for some pages in the (*addr*, *addr + len*) range might have been changed. If the **ukey_protect** subroutine is unsuccessful, the `errno` global variable might be set to one of the following values:

Item	Description
EPERM	The caller does not have sufficient authority to set a user-key on target memory.
ENOSYS	User-keys are not supported.
EINVAL	The *ukey* parameter is not valid, or the *addr* parameter is not a multiple of the page size as returned by the `sysconf` subroutine using the *_SC_PAGE_SIZE* value for the *Name* parameter.
EFAULT	Address is not valid for the address space of the process.

Related Information

The **ukey_getkey** Subroutine.

ulimit Subroutine

Purpose

Sets and gets user limits.

Library

Standard C Library (**libc.a**)

Syntax

The syntax for the **ulimit** subroutine when the *Command* parameter specifies a value of **GET_FSIZE** or **SET_FSIZE** is:

```
#include <ulimit.h>

long int ulimit ( Command,  NewLimit)
int Command;
off_t NewLimit;
```

The syntax for the **ulimit** subroutine when the *Command* parameter specifies a value of **GET_DATALIM**, **SET_DATALIM**, **GET_STACKLIM**, **SET_STACKLIM**, **GET_REALDIR**, or **SET_REALDIR** is:

```
#include <ulimit.h>
```

```
long int ulimit (Command, NewLimit)
int Command;
unsigned long NewLimit;
```

Description

The **ulimit** subroutine controls process limits.

Even with remote files, the **ulimit** subroutine values of the process on the client node are used.

Note: Raising the data ulimit does not necessarily raise the program break value. If the proper memory segments are not initialized at program load time, raising your memory limit will not allow access to this memory. Also, without these memory segments initialized, the value returned after such a change may not be the proper break value. If your data limit is RLIM_INFINITY, this value will never advance past the segment size, even if that data is available. Use the **-bmaxdata** flag of the **ld** command to set up these segments at load time.

Setting an fsize of 2G or more for a 32-bit application will be treated as unlimited.

Parameters

Item	Description
Command	Specifies the form of control. The following *Command* parameter values require that the *NewLimit* parameter be declared as an **off_t** structure: **GET_FSIZE (1)** Returns the process file size limit. The limit is in units of **UBSIZE** blocks (see the **sys/param.h** file) and is inherited by child processes. Files of any size can be read. The process file size limit is returned in the **off_t** structure specified by the *NewLimit* parameter. **SET_FSIZE (2)** Sets the process file size limit to the value in the **off_t** structure specified by the *NewLimit* parameter. Any process can decrease this limit, but only a process with root user authority can increase the limit. The new file size limit is returned. The following *Command* parameter values require that the *NewLimit* parameter be declared as an integer: **GET_DATALIM (3)** Returns the maximum possible break value (as described in the **brk** or **sbrk** subroutine). **SET_DATALIM (1004)** Sets the maximum possible break value (described in the **brk** and **sbrk** subroutines). Returns the new maximum break value, which is the *NewLimit* parameter rounded up to the nearest page boundary. **Note:** When a program is executing using the large address-space model, the operating system attempts to modify the soft limit on data size, if necessary, to increase it to match the *maxdata* value. If the *maxdata* value is larger than the current hard limit on data size, either the program will not execute if the XPG_SUS_ENV environment variable has the value set to ON, or the soft limit will be set to the current hard limit. If the *maxdata* value is smaller than the size of the program's static data, the program will not execute. **GET_STACKLIM (1005)** Returns the lowest valid stack address. **Note:** Stacks grow from high addresses to low addresses. **SET_STACKLIM (1006)** Sets the lowest valid stack address. Returns the new minimum valid stack address, which is the *NewLimit* parameter rounded down to the nearest page boundary. **GET_REALDIR (1007)** Returns the current value of the **real directory read** flag. If this flag is a value of 0, a **read** system call (or **readx** with *Extension* parameter value of 0) against a directory returns fixed-format entries compatible with the System V UNIX operating system. Otherwise, a **read** system call(or **readx** with *Extension* parameter value of 0) against a directory returns the underlying physical format. **SET_REALDIR (1008)** Sets the value of the **real directory read** flag. If the *NewLimit* parameter is a value of 0, this flag is cleared; otherwise, it is set. The old value of the **real directory read** flag is returned.
NewLimit	Specifies the new limit. The value and data type or structure of the *NewLimit* parameter depends on the *Command* parameter value that is used.

Examples

To increase the size of the stack by 4096 bytes (use 4096 or PAGESIZE), and set the `rc` to the new lowest valid stack address, enter:

```
rc = ulimit(SET_STACKLIM, ulimit(GET_STACKLIM, 0) - 4096);
```

Return Values

Upon successful completion, the value of the requested limit is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

All return values are permissible if the **ulimit** subroutine is successful. To check for error situations, an application should set the **errno** global variable to 0 before calling the **ulimit** subroutine. If the **ulimit** subroutine returns a value of -1, the application should check the **errno** global variable to verify that it is nonzero.

Error Codes

The **ulimit** subroutine is unsuccessful and the limit remains unchanged if one of the following is true:

Item	Description
EPERM	A process without root user authority attempts to increase the file size limit.
EINVAL	The *Command* parameter is a value other than **GET_FSIZE**, **SET_FSIZE**, **GET_DATALIM**, **SET_DATALIM**, **GET_STACKLIM**, **SET_STACKLIM**, **GET_REALDIR**, or **SET_REALDIR**.

umask Subroutine

Purpose

Sets and gets the value of the file creation mask.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/stat.h>
```

mode_t umask (*CreationMask*)
mode_t *CreationMask*;

Description

The **umask** subroutine sets the file-mode creation mask of the process to the value of the *CreationMask* parameter and returns the previous value of the mask.

Whenever a file is created (by the **open**, **mkdir**, or **mknod** subroutine), all file permission bits set in the file mode creation mask are cleared in the mode of the created file. This clearing allows users to restrict the default access to their files.

The mask is inherited by child processes.

Parameters

Item	Description
CreationMask	Specifies the value of the file mode creation mask. The *CreationMask* parameter is constructed by logically ORing file permission bits defined in the **sys/mode.h** file. Nine bits of the *CreationMask* parameter are significant.

Return Values

If successful, the file permission bits returned by the **umask** subroutine are the previous value of the file-mode creation mask. The *CreationMask* parameter can be set to this value in subsequent calls to the **umask** subroutine, returning the mask to its initial state.

umount or uvmount Subroutine

Purpose

Removes a virtual file system from the file tree.

Library

Standard C Library (**libc.a**)

Syntax

```
int umount ( Device)
char *Device;
```

```
#include <sys/vmount.h>
```

```
int uvmount ( VirtualFileSystemID,  Flag)
int VirtualFileSystemID;
int Flag;
```

Description

The **umount** and **uvmount** subroutines remove a virtual file system (VFS) from the file tree.

The **umount** subroutine unmounts only file systems mounted from a block device (a special file identified by its path to the block device).

In addition to local devices, the **uvmount** subroutine unmounts local or remote directories, identified by the *VirtualFileSystemID* parameter.

Only a calling process with root user authority or in the system group and having write access to the mount point can unmount a device, file and directory mount.

Parameters

Item	Description
Device	The path name of the block device to be unmounted for the **umount** subroutine.

Item	Description
VirtualFileSystemID	The unique identifier of the VFS to be unmounted for the **uvmount** subroutine. This value is returned when a VFS is created by the **vmount** subroutine and may subsequently be obtained by the **mntctl** subroutine. The *VirtualFileSystemID* is also reported in the **stat** subroutine `st_vfs` field.
Flag	Specifies special action for the **uvmount** subroutine. Currently only one value is defined: **UVMNT_FORCE** Force the unmount. This flag is ignored for device mounts.

Return Values

Upon successful completion a value of 0 is returned. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **uvmount** subroutine fails if one of the following is true:

Item	Description
EPERM	The calling process does not have write permission to the root of the VFS, the mounted object is a device or remote, and the calling process does not have root user authority.
EINVAL	No VFS with the specified *VirtualFileSystemID* parameter exists.
EBUSY	A device that is still in use is being unmounted.

The **umount** subroutine fails if one of the following is true:

Item	Description
EPERM	The calling process does not have root user authority.
ENOENT	The *Device* parameter does not exist.
ENOBLK	The *Device* parameter is not a block device.
EINVAL	The *Device* parameter is not mounted.
EINVAL	The *Device* parameter is not local.
EBUSY	A process is holding a reference to a file located on the file system.

The **umount** subroutine can be unsuccessful for other reasons. For a list of additional errors, see ../bostechref/bos_error_codes.dita.

uname or unamex Subroutine

Purpose

Gets the name of the current operating system.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/utsname.h>
int uname ( Name)
struct utsname *Name;
int unamex ( Name)
struct xutsname *Name;
```

Description

The **uname** subroutine stores information identifying the current system in the structure pointed to by the *Name* parameter.

The **uname** subroutine uses the **utsname** structure, which is defined in the **sys/utsname.h** file, and contains the following members:

```
char   sysname[SYS_NMLN];
char   nodename[SYS_NMLN];
char   release[SYS_NMLN];
char   version[SYS_NMLN];
char   machine[SYS_NMLN];
```

The **uname** subroutine returns a null-terminated character string naming the current system in the `sysname` character array. The `nodename` array contains the name that the system is known by on a communications network. The `release` and `version` arrays further identify the system. The `machine` array identifies the system unit hardware being used. The `utsname.machine` field is not unique if the last two characters in the string are `4C`. The character string returned by the **uname -Mu** command is unique for all systems and the character string returned by the **uname -MuL** command is unique for all partitions is all systems.

The **unamex** subroutine uses the **xutsname** structure, which is defined in the **sys/utsname.h** file, and contains the following members:

```
unsigned int   nid;
int   reserved;
unsigned long long   longnid;
```

The `xutsname.nid` field is the binary form of the `utsname.machine` field. The `xutsname.nid` field is not unique if the last two nibbles are `0x4C`. The character string returned by the **uname -Mu** command is unique for all systems and the character string returned by the **uname -MuL** command is unique for all partitions in all systems. For local area networks in which a binary node name is appropriate, the `xutsname.nid` field contains such a name.

Release and version variable numbers returned by the **uname** and **unamex** subroutines may change when new BOS software levels are installed. This change affects applications using these values to access licensed programs. Machine variable changes are due to hardware fixes or upgrades.

Contact the appropriate support organization if your application is affected.

Parameters

Item	Description
Name	A pointer to the **utsname** or **xutsname** structure.

Return Values

Upon successful completion, the **uname** or **unamex** subroutine returns a nonnegative value. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **uname** and **unamex** subroutines is unsuccessful if the following is true:

Item	Description
EFAULT	The *Name* parameter points outside of the process address space.

unctrl Subroutine

Purpose

Generates a printable representation of a character.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>

char *unctrl
(chtype c);
```

Description

The **unctrl** subroutine generates a character string that is a printable representation of *c*. If *c* is a control character, it is converted to the ^X notation. If *c* contains rendition information, the effect is undefined.

Parameters

Item	Description
c	

Return Values

Upon successful completion, the **unctrl** subroutine returns the generated string. Otherwise, it returns a null pointer.

Examples

To display a printable representation of the newline character, enter:

```
char *new_line;
int my_character;
addstr ("Hit the enter key.");
my_character=getch();
new_line=unctrl (my_character);
printw (Newline=%s", new_line);
refresh();
```

This prints, "newline=^J".

ungetc or ungetwc Subroutine

Purpose

Pushes a character back into the input stream.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdio.h>
```

```
int ungetc ( Character,  Stream)
int Character;
FILE *Stream;
```

```
wint_t ungetwc (Character, Stream)
wint_t Character;
FILE *Stream;
```

Description

The **ungetc** and **ungetwc** subroutines insert the character specified by the *Character* parameter (converted to an unsigned character in the case of the **ungetc** subroutine) into the buffer associated with the input stream specified by the *Stream* parameter. This causes the next call to the **getc** or **getwc** subroutine to return the *Character* value. A successful intervening call (with the stream specified by the *Stream* parameter) to a file-positioning subroutine (**fseek**, **fsetpos**, or **rewind**) discards any inserted characters for the stream. The **ungetc** and **ungetwc** subroutines return the *Character* value, and leaves the file (in its externally stored form) specified by the *Stream* parameter unchanged.

You can always push one character back onto a stream, provided that something has been read from the stream or the **setbuf** subroutine has been called. If the **ungetc** or **ungetwc** subroutine is called too many times on the same stream without an intervening read or file-positioning operation, the operation may not be successful. The **fseek** subroutine erases all memory of inserted characters.

The **ungetc** and **ungetwc** subroutines return a value of **EOF** or **WEOF** if a character cannot be inserted.

A successful call to the **ungetc** or **ungetwc** subroutine clears the end-of-file indicator for the stream specified by the *Stream* parameter. The value of the file-position indicator after all inserted characters are read or discarded is the same as before the characters were inserted. The value of the file-position indicator is decreased after each successful call to the **ungetc** or **ungetwc** subroutine. If its value was 0 before the call, its value is indeterminate after the call.

Parameters

Item	Description
Character	Specifies a character.
Stream	Specifies the input stream.

Return Values

The **ungetc** and **ungetwc** subroutines return the inserted character if successful; otherwise, **EOF** or **WEOF** is returned, respectively.

ungetch, unget_wch Subroutine

Purpose

Pushes a character onto the input queue.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int ungetch
(int ch);
```

```
int unget_wch
(const wchar_t wch);
```

Description

The **ungetch** subroutine pushes the single-byte character *ch* onto the head of the input queue.

The **unget_wch** subroutine pushes the wide character *wch* onto the head of the input queue.

One character of push-back is guaranteed. The result of successive calls without an intervening call to the **getch** or **get_wch** subroutine are unspecified.

Parameters

Item	Description
ch	
wch	

Examples

To force the key KEY_ENTER back into the queue, use:

```
ungetch(KEY_ENTER);
```

ulckpwdf Subroutine

Purpose

The **ulckpwdf** subroutine unlocks the password database file.

Library

Security Library (**libc.a**)

Syntax

```
#include <pwd.h>
int ulckpwdf ()
```

Description

The **ulckpwdf** subroutine releases the lock on a `/etc/security/.pwdlck` file that is held by using the **lckpwdf** routine.

Return Values

The **ulckpwdf** subroutine returns a value of **0** for success and a value of **-1** when the lock is already released.

unlink or unlinkat Subroutine

Purpose

Removes a directory entry.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <unistd.h>

int unlink (Path)
const char *Path;

int unlinkat (DirFileDescriptor,
Path, Flag)
int DirFileDescriptor;
const char * Path;
int Flag;
```

Description

The **unlink** and **unlinkat** subroutines remove the directory entry specified by the *Path* parameter and decrease the link count of the file referenced by the link. If Network File System (NFS) is installed on your system, this path can cross into another node.

Attention: Removing a link to a directory requires root user authority. Unlinking of directories is strongly discouraged since erroneous directory structures can result. The **rmdir** subroutine, or the **unlinkat** subroutine with the *Flag* parameter set to **AT_REMOVEDIR**, should be used to remove empty directories.

When all links to a file are removed and no process has the file open, all resources associated with the file are reclaimed, and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the directory entry disappears. However, the removal of the file contents is postponed until all references to the file are closed.

If the parent directory of *Path* has the **sticky** attribute (described in the **mode.h** file), the calling process must have root user authority or an effective user ID equal to the owner ID of *Path* or the owner ID of the parent directory of *Path*.

The `st_ctime` and `st_mtime` fields of the parent directory are marked for update if the **unlink** or **unlinkat** subroutine is successful. In addition, if the file's link count is not 0, the `st_ctime` field of the file will be marked for update.

Applications should use the **rmdir** subroutine, or the **unlinkat** subroutine with the *Flag* parameter having the **AT_REMOVEDIR** bit on, to remove a directory. If the *Path* parameter names a symbolic link, the link itself is removed.

The **unlinkat** subroutine is equivalent to the **unlink** subroutine if the *Flag* parameter does not have the **AT_REMOVEDIR** bit set, and if the *DirFileDescriptor* is **AT_FDCWD** or *Path* is an absolute path name. If *DirFileDescriptor* is a valid file descriptor of an open directory and *Path* is a relative path name, *Path* is considered to be relative to the directory that is associated with the *DirFileDescriptor* parameter instead of the current working directory.

If the *DirFileDescriptor* in the **unlinkat** subroutine was opened without the **O_SEARCH** open flag, the subroutine checks to determine whether directory searches are permitted for that directory by using the current permissions of the directory. If the directory was opened with the **O_SEARCH** open flag, the subroutine does not perform the check for that directory.

If the *Flag* parameter of the **unlinkat** subroutine has the **AT_REMOVEDIR** bit set, the **unlinkat** subroutine is equivalent to the **rmdir** subroutine.

Parameters

Item	Description
DirFileDescriptor	Specifies the file descriptor of an open directory.
Path	Specifies the directory entry to be removed. If *DirFileDescriptor* is specified and *Path* is a relative path name, then *Path* is considered relative to the directory specified by *DirFileDescriptor*.
Flag	Specifies a bit field. If it contains the **AT_REMOVEDIR** bit and Path points to a directory, then the directory specified by *Path* is removed.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, the **errno** global variable is set to indicate the error, and the specified file is not changed.

Error Codes

The **unlink** and **unlinkat** subroutines fail and the named file is not unlinked if one of the following is true:

Item	Description
ENOENT	The named file does not exist.
EACCES	Write permission is denied on the directory containing the link to be removed.
EBUSY	The entry to be unlinked is the mount point for a mounted file system, or the file named by *Path* is a named STREAM.
EPERM	The file specified by the *Path* parameter is a directory, and the calling process does not have root user authority. **EPERM** is also returned if the file named by the *Path* parameter is a directory in a JFS2 file system. Note that JFS allows you to unlink a directory.
EROFS	The entry to be unlinked is part of a read-only file system.

The **unlinkat** subroutine is unsuccessful if one or more of the following is true:

Item	Description
EBADF	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a valid file descriptor.
ENOTDIR	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.
EINVAL	The value of the *Flag* parameter is not valid.

The **unlink** and **unlinkat** subroutines can be unsuccessful for other reasons. For a list of additional errors, see ../bostechref/bos_error_codes.dita

If NFS is installed on the system, the **unlink** and **unlinkat** subroutines can also fail if the following is true:

Item	Description
ETIMEDOUT	The connection timed out.

unload and terminateAndUnload Subroutines

Purpose

Unloads a module.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/ldr.h>

int unload( FunctionPointer)
int (*FunctionPointer)( );

int terminateAndUnload( FunctionPointer)
int (*FunctionPointer)( );
```

Description

The **unload** and `terminateAndUnload` subroutines unload the specified module and its dependents. The value returned by the **load** subroutine is passed to the **unload** subroutine as *FunctionPointer*. The **unload** subroutine calls termination routines (fini routines) for the specified module and any of its dependents that are not being used by any other module.

The **unload** and `terminateAndUnload` subroutines free the storage used by the specified module only if the module is no longer in use. A module is in use as long as any other module that is in use imports symbols from it.

The **unload** subroutine does not perform C++ termination, that is, calling destructors. Use the **terminateAndUnload** subroutine instead. The `dlclose` subroutine performs C++ termination like the **terminateAndUnload** subroutine does.

When a module is unloaded, any deferred resolution symbols that were bound to the module remain bound. These bindings create references to the module that cannot be undone, even with the **unload** subroutine.

When a process executing under **ptrace** control calls **unload**, the debugger is notified by setting the **W_SLWTED** flag in the status returned by **wait**. If a module loaded in the shared library is no longer in use by the process, the module is deleted from the process's copy of the shared library segment by freeing the pages containing the module.

Parameters

Item	Description
FunctionPointer	Specifies the name of the function that returns.

Return Values

Upon successful completion, the **unload** and `terminateAndUnload` subroutines return a value of `0`, even if the module couldn't be unloaded because it is still in use.

Error Codes

If the **unload** and `terminateAndUnload` subroutines fail, a value of `-1` is returned, the program is not unloaded, and **errno** is set to indicate the error. **errno** may be set to one of the following:

Item	Description
EINVAL	The *FunctionPointer* parameter does not correspond to a program loaded by the **load** subroutine.

unlockpt Subroutine

Purpose

Unlocks a pseudo-terminal device.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

int unlockpt ( FileDescriptor)
int FileDescriptor;
```

Description

The **unlockpt** subroutine unlocks the worker peudo-terminal device associated with the master peudo-terminal device defined by the *FileDescriptor* parameter. This subroutine has no effect if the environment variable XPG_SUS_ENV is not set equal to the string "ON", or if the BSD PTY driver is used.

Parameters

Item	Description
FileDescriptor	Specifies the file descriptor of the master pseudo-terminal device.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

usrinfo Subroutine

Purpose

Gets and sets user information about the owner of the current process.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <uinfo.h>

int usrinfo ( Command,  Buffer,  Count)
int Command;
char *Buffer;
int Count;
```

Description

The **usrinfo** subroutine gets and sets information about the owner of the current process. The information is a sequence of null-terminated *name*=*value* strings. The last string in the sequence is terminated by two successive null characters. A child process inherits the user information of the parent process.

Parameters

Item	Description
Command	Specifies one of the following constants: **GETUINFO** Copies user information, up to the number of bytes specified by the *Count* parameter, into the buffer pointed to by the *Buffer* parameter. **SETUINFO** Sets the user information for the process to the number of bytes specified by the *Count* parameter in the buffer pointed to by the *Buffer* parameter. The calling process must have root user authority to set the user information. The minimum user information consists of four strings typically set by the **login** program: **NAME=*UserName*** **LOGIN=*LoginName*** **LOGNAME=*LoginName*** **TTY=*TTYName*** If the process has no terminal, the *TTYName* parameter should be null.
Buffer	Specifies a pointer to a user buffer. This buffer is usually **UINFOSIZ** bytes long.
Count	Specifies the number of bytes of user information copied from or to the user buffer.

Return Values

If successful, the **usrinfo** subroutine returns a non-negative integer giving the number of bytes transferred. Otherwise, a value of -1 is returned and the **errno** global variable is set to indicate the error.

Error Codes

The **usrinfo** subroutine fails if one of the following is true:

Item	Description
EPERM	The *Command* parameter is set to **SETUINFO**, and the calling process does not have root user authority.
EINVAL	The *Command* parameter is not set to **SETUINFO** or **GETUINFO**.
EINVAL	The *Command* parameter is set to **SETUINFO**, and the *Count* parameter is larger than **UINFOSIZ**.
EFAULT	The *Buffer* parameter points outside of the address space of the process.

utime, utimes, futimens, or utimensat Subroutine

Purpose

Sets file-access and modification times.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/time.h>

int utimes ( Path,  Times)
char *Path;
struct timeval Times[2];

#include <utime.h>

int utime ( Path,  Times)
const char *Path;
const struct utimbuf *Times;

#include <sys/stat.h>
int futimens ( FileDescriptor, Times)
int FileDescriptor;
struct timespec Times[2];

#include <sys/stat.h>
int utimensat ( DirFileDescriptor, Path, Times, Flag)
int DirFileDescriptor;
char *Path;
struct timespec Times[2];
int Flag;
```

Description

The **utimes** subroutine sets the access and modification times of the file pointed to by the *Path* parameter to the value of the *Times* parameter.

The **futimens** and **utimensat** subroutines set the access and modification times of a file to the value of the *Times* parameter. The **futimens** subroutine file is specified by the *FileDescriptor* parameter, which is a file descriptor of an open file.

The **utimensat** subroutine file is specified by the *DirFileDescriptor* and *Path* parameters. If the *DirFileDescriptor* parameter is set to **AT_FDCWD** or the *Path* parameter is an absolute path name, the **utimensat** subroutine is equivalent to the **utimes** subroutine when the *Flag* parameter of the **utimensat** subroutine set to zero.

If the **tv_nsec** field of a **timespec** structure or the **tv_usec** field of the **timeval** structure has the value **UTIME_NOW**, the corresponding timestamp for the file is set to the current time. If the field has the value **UTIME_OMIT**, the corresponding timestamp for the file is not changed. In either case, the **tv_sec** field is ignored. If the *Times* parameter is **NULL**, both timestamps are set to the current time.

The **utime** subroutine also sets file access and modification times. Each time is contained in a single integer and is accurate only to the nearest second. If successful, the **utime** subroutine marks the time of the last file-status change (**st_ctime**) to be updated.

Parameters

Item	Description
FileDescriptor	Specifies the file descriptor of an open file.
Path	Points to the path name of the file. If the *DirFileDescriptor* is specified and Path is a relative path name, then Path is considered relative to the directory specified by *DirFileDescriptor*.
DirFileDescriptor	Specifies the file descriptor of an open directory.
Flag	Specifies a bit field. If the **AT_SYMLINK_NOFOLLOW** bit is set and Path points to a symbolic link, then the access and modification times of the symbolic link are changed. If the **AT_SYMLINK_NOFOLLOW** bit is not set, the times of the file the symbolic link points at are changed.

Item	Description
Times	Specifies the date and time of last access and of last modification. For the **utimes** subroutine, this is an array of **timespec** structures, as defined in the **<sys/time.h>** file. The first array element represents the date and time of last access, and the second element represents the date and time of last modification. The times in the **timeval** structure are measured in seconds and microseconds since 00:00:00 Greenwich Mean Time (GMT), 1 January 1970. The times in the **timespec** structure are measured in seconds and nanoseconds since 00:00:00 Greenwich Mean Time (GMT), 1 January 1970. For the **utime** subroutine, this parameter is a pointer to a **utimbuf** structure, as defined in the **utime.h** file. The first structure member represents the date and time of last access, and the second member represents the date and time of last modification. The times in the **utimbuf** structure are measured in seconds since 00:00:00 Greenwich Mean Time (GMT), 1 January 1970. If the *Times* parameter has a null value, the access and modification times of the file are set to the current time. If the file is remote, the current time at the remote node, rather than the local node, is used. To use the call this way, the effective user ID of the process must be the same as the owner of the file or must have root authority, or the process must have write permission to the file. If the Times parameter does not have a null value and the **UTIME_NOW** or the **UTIME_OMIT** values are not set, the access and modification times are set to the values contained in the designated structure, regardless of whether those times are the same as the current time. Only the owner of the file or a user with root authority can use the call this way.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, the **errno** global variable is set to indicate the error, and the file times are not changed.

Error Codes

The **futimens**, **utimensat**, **utimes** or **utime** subroutines fail if one of the following is true:

Item	Description
EPERM	The *Times* parameter is not null and the calling process neither owns the file nor has root user authority.
EACCES	The *Times* parameter is null, effective user ID is neither the owner of the file nor has root authority, or write access is denied.
EROFS	The file system that contains the file is mounted read-only.

The **utimensat** subroutine fails if one or more of the following settings are true:

Item	Description
EBADF	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a valid file descriptor.
ENOTDIR	The *Path* parameter does not specify an absolute path and the *DirFileDescriptor* parameter is neither **AT_FDCWD** nor a file descriptor associated with a directory.
EINVAL	The value of the *Flag* parameter is not valid.

The **futimens** subroutine fails if the following is true:

Item	Description
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor.

The **futimens** or **utimensat** subroutine fails if one or more of the following settings are true:

Item	Description
EINVAL	The *Times* parameter has a negative **tv_sec** field.
EINVAL	The *Times* parameter has a negative **tv_nsec** field, or the **tv_nsec** field is equal to or greater than 1000 million.

The **utimes** subroutine fails if one or more of the following settings are true:

Item	Description
EINVAL	The *Times* parameter has a negative **tv_sec** field.
EINVAL	The *Times* parameter has a negative **tv_usec** field, or the **tv_usec** field is greater than or equal to a million.

The **utimes**, **utimensat**, or **utime** subroutine can be unsuccessful for other reasons. For a list of additional errors, see ../bostechref/bos_error_codes.dita

uuid_create or uuid_create_nil Subroutine

Purpose

Creates a universally unique identifier (UUID).

Libary

Standard C Library (**libc.a**)

Syntax

```
#include <uuid.h>

void uuid_create (uuid, status )
void uuid_create_nil (uuid, status)
uuid_t *uuid;
unsigned32 *status
```

Description

The **uuid_create** subroutine creates a new binary UUID, and stores it in the location pointed to by uuid. In case of success, the **uuid_create_nil** subroutine will set the location pointed to by the status to uuid_s_ok. The uuid parameter must not be NULL and point to a valid location.

Parameters

Item	Description
uuid	Points to the location where the universally unique identifier will be stored.
status	Points to the location where the status of uuid_create_nil will be stored.

Return Values

If successful, a value of 1 is returned. If unsuccessful, a value of -1 is returned.

uuid_hash Subroutine

Purpose

Creates a hash value for a given universally unique identifier (UUID).

Libary

Standard C Library (**libc.a**)

Syntax

```
#include <uuid.h>

unsigned16 uuid_hash(uuid, status )
uuid_p_t              uuid;
unsigned32           *status;
```

Description

The **uuid_hash** subroutine returns a 16-bit hash value for a given UUID.

Parameters

Item	Description
uuid	Points to the UUID for which the hash is to be generated
status	Points to the location to store the status of the operation

Return Values

The **uuid_hash** subroutine returns a 16-bit hash value

uuid_is_nil, uuid_compare, or uuid_equal Subroutine

Purpose

Compares universally unique identifiers (UUIDs).

Libary

Standard C Library (**libc.a**)

Syntax

```
#include <uuid.h>

signed32 (uuid1, uuid2, status )
boolean32     uuid_equa (uuid1, uuid2, status)
boolean32     uuid_is_nil (uuid1, status)
uuid_p_t      uuid1;
```

```
uuid_p_t        uuid2;
unsigned32      status;
```

Description

The **uuid_is_nil** subroutine checks whether the binary UUID pointed to by uuid1 is a nil UUID. The **uuid_compare** subroutine compares two binary UUIDs. The **uuid_equal** subroutine checks if two binary UUIDs are equal. If either of the parameters is a NULL pointer, the other parameter will be compared against the nil UUID.

Parameters

Item	Description
uuid1	Pointer identifying the first UUID to be compared
uuid2	Pointer identifying the second UUID to be compared
status	Points to the location where the status of the operation is to be stored.

Return Values

The **uuid_is_nil** subroutine returns a 1 if the UUID passed is a nil UUID, otherwise it returns 0. The **uuid_equal** subroutine returns a 1 if both UUIDs are equal, otherwise it returns 0. The **uuid_compare** subroutine returns a -1 if the uuid1 is lexically before uuid2, returns 0 if both the uuid1 and uuid2 are equal, otherwise the **uuid_compare** subroutine returns a -1.

uuid_to_string or uuid_from_string Subroutine

Purpose

Convert between binary and string universally unique identifiers (UUIDs).

Libary

Standard C Library (**libc.a**)

Syntax

```
#include <uuid.h>

void uuid_to_string(uuid, uuid_string, status )
void uuid_from_string(uuid_string,uuid, status)
uuid_p_t            uuid;
unsigned_char_p_t  *uuid_string;
unsigned32         *status;
```

Description

The **uuid_to_string** subroutine converts a binary UUID to a string UUID. The uuid_string parameter should point to an area of memory with enough space to store the string UUID, otherwise the results are undefined. If a NULL value is passed as the second argument of the uuid_to_string parameter, the required memory will be automatically allocated by calling the **malloc** subroutine. The **uuid_from_string** subroutine converts a string UUID to a binary UUID. The length of the string passed to the uuid_from_string parameter should be 0 or the length of UUID_C_UUID_STRING_MAX . On successful completion, uuid_s_ok is stored in the location pointed to by the status parameter.

Parameters

Item	Description
uuid	Points to the location containing the binary UUID
uuid_string	Points to the location containing the string UUID
status	Points to the location where the status of the operation is stored

Return Values

There are no return values, however, in case the string passed to the **uuid_from_string** subroutine is invalid, the location pointed to by the status parameter is set to uuid_s_invalid_string_uuid. If a NULL value is passed to the **uuid_to_string** subroutine as the second parameter and the system has run out of memory, the location pointed to by the status parameter is set to uuid_s_no_memory.

V

The following Base Operating System (BOS) runtime services begin with the letter *v*.

varargs Macros

Purpose

Handles a variable-length parameter list.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdarg.h>

type va_arg ( Argp,  Type)
va_list Argp;

void va_start (Argp,  ParmN)
va_list Argp;

void va_end (Argp)
va_list Argp;
```

OR

```
#include <varargs.h>

va_alist Argp;
va_dcl

void va_start (Argp)
va_list Argp;

type va_arg (Argp, Type)
va_list Argp;

void va_end (Argp)
va_list Argp;
```

Description

The **varargs** set of macros allows you to write portable subroutines that accept a variable number of parameters. Subroutines that have variable-length parameter lists (such as the **printf** subroutine), but that do not use the **varargs** macros, are inherently nonportable because different systems use different parameter-passing conventions.

Note: Do not include both **<stdarg.h>** and **<varargs.h>**. Use of **<varargs.h>** is not recommended. It is supplied for backwards compatibility.

For <stdarg.h>

Item	Description
va_start	Initializes the *Argp* parameter to point to the beginning of the list. The *ParmN* parameter identifies the rightmost parameter in the function definition. For compatibility with previous programs, it defaults to the address of the first parameter on the parameter list. Acceptable parameters include: integer, double, and pointer. The **va_start** macro is started before any access to the unnamed arguments.

For <varargs.h>

Item	Description
va_alist	A variable used as the parameter list in the function header.
va_argp	A variable that the **varargs** macros use to keep track of the current location in the parameter list. Do not modify this variable.
va_dcl	Declaration for **va_alist**. No semicolon should follow **va_dcl**.
va_start	Initializes the *Argp* parameter to point to the beginning of the list.

For <stdarg.h> and <varargs.h>

Item	Description
va_list	Defines the type of the variable used to traverse the list.
va_arg	Returns the next parameter in the list pointed to by the *Argp* parameter.
va_end	Cleans up at the end.

Your subroutine can traverse, or scan, the parameter list more than once. Start each traversal with a call to the **va_start** macro and end it with the **va_end** macro.

Note: The calling routine is responsible for specifying the number of parameters because it is not always possible to determine this from the stack frame. For example, **execl** is passed a null pointer to signal the end of the list. The **printf** subroutine determines the number of parameters from its *Format* parameter.

Parameters

Item	Description
Argp	Specifies a variable that the **varargs** macros use to keep track of the current location in the parameter list. Do not modify this variable.
Type	Specifies the type to which the expected argument will be converted when passed as an argument. In C, arguments that are char or short should be accessed as int; unsigned char or short arguments are converted to unsigned int, and float arguments are converted to double. Different types can be mixed, but it is up to the routine to know what type of argument is expected, because it cannot be determined at runtime.
ParmN	Specifies a parameter that is the identifier of the rightmost parameter in the function definition.

Examples

The following **execl** system call implementations are examples of the **varargs** macros usage.

1. The following example includes **<stdarg.h>**:

```
#include <stdarg.h>
#define MAXargs 31
int execl (const char *path, ...)
{
   va_list Argp;
   char *array [MAXargs];
   int argno=0;
```

```
    va_start (Argp, path);
    while ((array[argno++] = va_arg(Argp, char*)) != (char*)0)
            ;
    va_end(Argp);
    return(execv(path, array));

}
main()
{
    execl("/usr/bin/echo", "ArgV[0]", "This", "Is", "A", "Test",      "\0");
    /* ArguementV[0] will be discarded by the execv in main(): */
    /* by convention ArgV[0] should be a copy of path parameter */
}
```

2. The following example includes **<varargs.h>**:

```
#include <varargs.h>
#define MAXargS 100
/*
**  execl is called by
**  execl(file, arg1, arg2, . . . , (char *) 0);
*/
execl(va_alist)
    va_dcl
{   va_list ap;
    char *file;
    char *args[MAXargS];
    int argno = 0;
    va_start(ap);
    file = va_arg(ap, char *);
    while ((args[argno++] = va_arg(ap, char *)) != (char *) 0)
       ;   /* Empty loop body */
    va_end(ap);
    return (execv(file, args));
}
```

vfscanf, vscanf, or vsscanf Subroutine

Upon successful completion, these functions shall return the number of successfully matched and assigned input items; this number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF shall be returned. If a read error occurs, the error indicator for the stream is set, EOF shall be returned, and *errno* shall be set to indicate the error.

Purpose

Formats input of an argument list.

Syntax

```
#include <stdarg.h>
#include <stdio.h>

int vfscanf (stream, format, arg)
File *restrict stream
const char  format;
va_list arg;

int vscanf (format, arg)
const char  format;
va_list arg;

int vsscanf (format, arg)
const char  format;
va_list arg;
```

Description

The **vscanf**, **vfscanf**, and **vsscanf** subroutines are equivalent to the **scanf**, **fscanf**, and **sscanf** subroutines, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the **<stdarg.h>** header file. These subroutines do not

invoke the **va_end** macro. As these functions invoke the **va_arg** macro, the value of *ap* after the return is unspecified.

Parameters

Item	Description
stream	
format	
arg	

Return Values

vfwscanf, vswscanf, or vwscanf Subroutine

Purpose

Wide-character formatted input of the argument list.

Syntax

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwscanf (stream, format, arg)
FILE *restrict stream;
const wchar_t format;
va_list arg;

int vswscanf (ws, format, arg)
const wchar_t *restrict ws;
const wchar_t format;
va_list arg;

int vwscanf (format, arg)
const wchar_t format;
va_list arg;
```

Description

The **vfwscanf**, **vswscanf**, and **vwscanf** subroutines are equivalent to the **fwscanf**, **swscanf**, and **wscanf** subroutines, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in the **<stdarg.h>** header file. These subroutines do not invoke the **va_end** macro. As these subroutines invoke the **va_arg** macro, the value of *ap* after the return is unspecified.

Return Values

Upon successful completion, the **vfwscanf**, **vswscanf**, and **vwscanf** subroutines return the number of successfully matched and assigned input items. This number can be zero in the event of an early matching failure. If the input ends before the first matching failure or conversion, EOF is returned. If a read error occurs, the error indicator for the stream is set, EOF is returned, and the **errno** global variable is set to indicate the error.

vfwprintf, vwprintf Subroutine

Purpose

Wide-character formatted output of a stdarg argument list.

Library

Standard library (**libc.a**)

Syntax

```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
```

```
int vwprintf ((const wchar_t * format, va_list arg) ;
int vfwprintf(FILE * stream, const wchar_t * format, va_list arg);
int vswprintf (wchar_t * s, size_t n, const wchar_t * format, va_list arg);
```

Description

The **vwprintf**, **vfwprintf** and **vswprintf** functions are the same as **wprintf**, **fwprintf** and **swprintf** respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined by **stdarg.h**.

These functions do not invoke the **va_end** macro. However, as these functions do invoke the **va_arg** macro, the value of **ap** after the return is indeterminate.

Return Values

Refer to **fwprintf**.

Error Codes

Refer to **fwprintf**.

vidattr, vid_attr, vidputs, or vid_puts Subroutine

Purpose

Outputs attributes to the terminal.

Library

Curses Library (**libcurses.a**)

Syntax

```
#include <curses.h>
```

```
int vidattr
(chtype attr);

int vid_attr
(attr_t attr,
short color_pair_number,
void *opt);
```

```
int vidputs
(chtype attr,
int (*putfunc)(int));
```

```
int vid_puts
(attr_t attr,
short color_pair_number,
void *opt,
int (*putfunc)(int));
```

Description

These subroutines output commands to a terminal that changes the terminal's attributes.

If the **terminfo** database indicates that the terminal in use can display characters in the rendition specified by *attr*, then the **vadattr** subroutine outputs one or more commands to request that the terminal display subsequent characters in that rendition. The subroutine outputs by calling the **putcha**r subroutine. The **vidattr** subroutine neither relies on nor updates the model that Curses maintains of the prior rendition mode.

The **vidputs** subroutine computes the same terminal output string that **vidattr** does, based on *attr*, but the **vidputs** subroutine outputs by calling the user-supplied subroutine **putfunc**. The **vid_attr** and **vid_puts** subroutines correspond to **vidattr** and **vidputs** respectively, but take a set of arguments, one of type *attr_t* for the attributes, *short* for the color pair number and a *void **, and thus support the attribute constants with the WA_prefix.

The *opts* argument is reserved for definition in a future edition of this document. Currently, the application must provide a null pointer as opts.

The user-supplied **putfunc** subroutine (which can be specified as an argument to either **vidputs** or **vid_puts** is either **putchar** or some other subroutine with the same prototype. Both the **vidputs** and the **vid_puts** subroutines ignore the return value of **putfunc**.

Parameters

Item	Description
att	
color_pair_number	
**opt*	
**putfunc*	

Return Values

Upon successful completion, these subroutines return OK. Otherwise, they return ERR.

Examples

1. To output the string that puts the terminal in its best standout mode through the **putchar** subroutine, enter

   ```
   vidattr(A_STANDOUT);
   ```

2. To output the string that puts the terminal in its best standout mode through the **putchar**-like subroutine `my_putc`, enter

   ```
   int (*my_putc) ();
   vidputs(A_STANDOUT, my_putc);
   ```

vmgetinfo Subroutine

Purpose

Retrieves Virtual Memory Manager (VMM) information.

Library

Standard C Library (**libc.a**)

Syntax

#include <sys/vminfo.h>

int vmgetinfo(void *out, int command, int arg)

Description

The **vmgetinfo** subroutine returns the current value of certain VMM parameters.

Parameters

arg
Additional parameter which depends on the command parameter.

command
Specifies which information is returned. The *command* parameter has the following valid values:

VMINFO
Returns the content of the **vminfo** structure (described in the **sys/vminfo.h** file). The *out* parameter points to a **vminfo** structure and the *arg* parameter is the size of the **vminfo** structure. The smaller value of the *arg* parameter and *sizeof* (**struct vminfo**) is copied.

VMINFO64
Returns the contents of the **vminfo64** structure (described in the **sys/vminfo.h** file). The *out* parameter points to a **vminfo64** structure and the *arg* parameter is the size of the **vminfo64** structure. The smaller value of the *arg* parameter and *sizeof* (**struct vminfo64**) is copied.

VMINFO_ABRIDGED
Returns the content of the **vminfo** structure (described in the **sys/vminfo.h** file). The **VMINFO_ABRIDGED** command updates only the non-time consuming statistics, therefore you must use the **VMINFO_ABRIDGED** command rather than the **VMINFO** command in performance-critical applications. The *out* parameter points to a **vminfo** structure and the *arg* parameter is the size of the **vminfo** structure. The smaller value between the *arg* and *sizeof* (**struct vminfo**) parameters is copied.

VM_PAGE_INFO
Returns the size, in bytes, of the page backing the address specified in the *addr* field of the **vm_page_info** structure (described in **sys/vminfo.h**). The *out* parameter points to a **vm_page_info** structure with the *addr* field set to the desired address of which to query the page size. The *arg* parameter is the size of the **vm_page_info** structure.

VM_NEW_HEAP_PSIZE
Sets a new preferred page size for future **sbreak** allocations for the calling process's private data heap. This page size setting is advisory. The *out* parameter is a pointer to a **psize_t** structure that contains the preferred page size, in bytes, to use to back any future **sbreak** allocations by the calling process. Presently, only 16M (0x1000000) and 4K (0x1000) are supported. The *arg* parameter is that of the *sizeof*(**psize_t**).

VM_SRAD_MEMINFO
Reports memory statistics about an Scheduler Resource Allocation Domain (SRAD). The *arg* parameter must contain the SRAD ID to be queried. The *out* parameter must be the pointer to

the **vm_srad_meminfo** structure whose first field, **vmsrad_in_size**, must contain the size of the structure.

The output of this command is stored in the following fields of the **vm_srad_meminfo** structure:

vmsrad_out_size
The number of bytes returned in the buffer.

vmsrad_total_pg
The total number of bytes of pageable memory contained in the SRAD. This value excludes the memory that is permanently reserved for low-level memory management.

Note: This field was formerly known as **vmsrad_total**.

vmsrad_free_pg
The amount of free pageable memory displayed in bytes in the SRAD.

Note: This field was formerly known as **vmsrad_free**.

vmsrad_total_nonpg
The total number of bytes of non-pageable memory contained in the SRAD.

vmsrad_free_nonpg
The amount of free non-pageable memory displayed in bytes in the SRAD.

vmsrad_file
The number of bytes occupied by files.

vmsrad_aff_priv_pct
This is the maximum percentage of private memory that is allocated from the specified SRAD by the default page placement algorithm based on the tunable **enhanced_affinity_private**, the SRAD's computational usage, and the SRAD's memory-to-CPU capacity ratio.

vmsrad_aff_avail_pct
The percentage of available computational memory that is remaining in the **vmpool** parameter based on the tunable **enhanced_affinity_vmpool_limit** parameter and the average system computational percentage.

The total number of computational memory bytes available in SRAD is the value in the **vmsrad_total** parameter, minus the sum of the values in the **vmsrad_free** and **vmsrad_file** parameters.

VM_STAGGER_DATA
Staggers the calling process's current **sbreak** value by a cumulative per-MCM stagger value. This stagger value must be set through the **vmo** option **data_stagger_interval**. The value of the *out* parameter is NULL and that of the *arg* parameter is 0.

IPC_LIMITS
Returns the content of the **ipc_limits** struct (described in the **sys/vminfo.h** file). The *out* parameter points to an **ipc_limits** structure and *arg* is the size of this structure. The smaller value of the *arg* and *sizeof* (struct **ipc_limits**) parameters is copied. The **ipc_limits** struct contains the inter-process communication (IPC) limits for the system.

VMINFO_GETPSIZES
Reports a system's supported page sizes. When the value of *arg* is set to 0, the *out* parameter is ignored, and the number of supported page sizes is returned. When the value of *arg* is greater than 0, the *arg* parameter value indicates the number of page sizes to report, and the *out* parameter must be a pointer to an array with the number of **psize_t** structures specified by the *arg* parameter. The array of the **psize_t** structure is updated with the system's supported page sizes in sorted order starting with the smallest supported page size. The number of array entries updated with page sizes is returned.

VMINFO_PSIZE
Reports detailed VMM statistics for a specified page size. The *out* parameter points to a **vminfo_psize** structure with the **psize** field set to a page size, in bytes, for which to return statistics. Set the value of the *arg* parameter to the size of the **vminfo_psize** structure.

VM_PROC_PF_INFO or VM_THREAD_PF_INFO
Returns the time taken for processing page faults caused by a process or thread. The total number of page faults is also returned. The **arg** parameter must contain the size of the vm_pf_info structure and the **out** parameter must contain a pointer to the vm_pf_info structure. The first three fields of the vm_pf_info structure (version, flags, and id) are input fields that are populated by the calling process. The output of this command is stored by vmgetinfo in the output fields of the vm_pf_info structure. The vm_pf_info structure is defined in sys/vminfo.h as follows:

```
struct vm_pf_info
{
       /* INPUT */
       uint32_t version;
       uint32_t flags; /* currently unused */
       id64_t id; /* pid or tid */
       /* OUTPUT */
       struct timestruc64_t text_major_pf_time;
       struct timestruc64_t data_major_pf_time;
       struct timestruc64_t kernel_major_pf_time;
       struct timestruc64_t text_minor_pf_time;
       struct timestruc64_t data_minor_pf_time;
       struct timestruc64_t kernel_minor_pf_time;
       uint64_t minor_pf_count;
       uint64_t major_pf_count;
}
```

The fields of the vm_pf_info structure follows:

version
Must be set to the VM_PF_INFO_VER value (defined in the sys/vminfo.h header file).

flags
Unused. Must be set to 0.

id
Must contain a valid thread or a process identifier(depending on the command), or a value of -1. If the value is -1, the information about the calling thread or the process is requested.

out
Specifies the address where VMM information is returned.

Return Values

For all commands other than **VMINFO_GETPSIZES**, 0 is returned if the **vmgetinfo** subroutine is successful. When **VMINFO_GETPSIZES** is specified as the command, a number of page sizes is returned if the **vmgetinfo** subroutine is successful.

If the **vmgetinfo** subroutine is unsuccessful, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **vmgetinfo** subroutine does not succeed if the following are true:

EFAULT
The copy operation to the buffer was not successful.

EFAULT
Attempt at reading the page size pointed to by the *out* parameter was not successful.

EINVAL
When **VM_PAGE_INFO** is the command, the *addr* field of the **vm_page_info** structure is an invalid address.

EINVAL
When **VM_NEW_HEAP_PSIZE** is the command, the *arg* parameter is not set to the size of **psize_t**.

EINVAL
When **VM_STAGGER_DATA** is the command, the *out* parameter is not set to NULL, or the *arg* parameter is not set to 0.

EINVAL
When **VMINFO_PSIZE** is the command, the **psize** field of the **vminfo_psize** structure is an unsupported page size, the *arg* parameter is less than the size of a **psize_t**, or the *out* parameter is NULL.

EINVAL
When **VMINFO_GETPSIZES** is the command, the *arg* parameter is less than 0, or the *out* parameter is NULL when the *arg* parameter is non-zero.

ENOMEM
When **VM_STAGGER_DATA** is the command, the calling process's data could not be staggered because of resource limitations on the process's data size. (Use **ulimit data** to increase the allowed data for this process. See the “ulimit Subroutine” on page 2254.)

ENOMEM
When **VM_NEW_HEAP_PSIZE** is the command, the break value of the process could not be adjusted because of resource limitations. (See the “ulimit Subroutine” on page 2254.)

ENOSYS
The *command* parameter is not valid (or not yet implemented).

ENOSYS
Not implemented in current version of AIX (or on 32-bit kernel).

ENOTSUP
When **VM_NEW_HEAP_PSIZE** is the command, the calling process is not 64-bit.

ENOTSUP
When **VM_STAGGER_DATA** is the command, the calling process is not 64-bit.

EPERM
When **VM_NEW_HEAP_PSIZE** is the command, the user does not have permission to use the requested page size.

ESRCH
When **VM_PROC_PF_INFO** or **VM_THREAD_PF_INFO** is the command, the thread or process identifier does not match any active thread or process.

EPERM
When you use the **VM_PROC_PF_INFO** or **VM_THREAD_PF_INFO** command, the user does not have sufficient role based access control (RBAC) privileges to retrieve information about the target process or thread.

EINVAL
When you use the **VM_PROC_PF_INFO or VM_THREAD_PF_INFO** , the version that is specified in the `vm_pf_info` structure does not match the **VM_PF_INFO_VER** value as seen by the `vmgetinfo` subroutine or, the flags field is not set to 0.

Examples

The following example demonstrates how an application could determine a system's supported page sizes with the **vmgetinfo()** subroutine:

```
int num_psizes;
psize_t *psizes;

/* Determine the number of supported page sizes */
num_psizes = vmgetinfo(NULL, VMINFO_GETPSIZES, 0);

if ((psizes = malloc(num_psizes*sizeof(psize_t))) == NULL)
    return(1);

/* Get the page sizes */
if (vmgetinfo(psizes, VMINFO_GETPSIZES, num_psizes)!= num_psizes)
{
```

```
    perror("vmgetinfo() unexpectedly failed");
    return(2);
}


/* psize[0] = smallest page size
 * psize[1] = next smallest page size...
 * psize[num_psizes-1] = largest supported page size
 */
```

vmount or mount Subroutine

Purpose

Makes a file system available for use.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/types.h>
#include <sys/vmount.h>
```

```
int vmount ( VMount,  Size)
struct vmount *VMount;
int Size;
```

```
int mount
( Device,  Path,  Flags)
char *Device;
char *Path;
int Flags;
```

Description

The **vmount** subroutine mounts a file system, thereby making the file available for use. The **vmount** subroutine effectively creates what is known as a *virtual file system*. After a file system is mounted, references to the path name that is to be mounted over refer to the root directory on the mounted file system.

A directory can only be mounted over a directory, and a file can only be mounted over a file. (The file or directory may be a symbolic link.)

Therefore, the **vmount** subroutine can provide the following types of mounts:

- A local file over a local or remote file
- A local directory over a local or remote directory
- A remote file over a local or remote file
- A remote directory over a local or remote directory.

A mount to a directory or a file can be issued if the calling process has root user authority or is in the system group and has write access to the mount point.

To mount a block device, remote file, or remote directory, the calling process must also have root user authority.

The **mount** subroutine only allows mounts of a block device over a local directory with the default file system type. The **mount** subroutine searches the **/etc/filesystems** file to find a corresponding stanza for the desired file system.

Note: The **mount** subroutine interface is provided only for compatibility with previous releases of the operating system. The use of the **mount** subroutine is strongly discouraged by normal application programs.

If the directory you are trying to mount over has the sticky bit set to on, you must either own that directory or be the root user for the mount to succeed. This restriction applies only to directory-over-directory mounts.

Parameters

Device
A path name identifying the block device (also called a special file) that contains the physical file system.

Path
A path name identifying the directory on which the file system is to be mounted.

Flags
Values that define characteristics of the object to be mounted. Currently these values are defined in the **/usr/include/sys/vmount.h** file:

MNT_READONLY
Indicates that the object to be mounted is read-only and that write access is not allowed. If this value is not specified, writing is permitted according to individual file accessibility.

MNT_NOSUID
Indicates that **setuid** and **setgid** programs referenced through the mount should not be executable. If this value is not specified, **setuid** and **setgid** programs referenced through the mount may be executable.

MNT_NODEV
Indicates that opens of device special files referenced through the mount should not succeed. If this value is not specified, opens of device special files referenced through the mount may succeed.

VMount
A pointer to a variable-length **vmount** structure. This structure is defined in the **sys/vmount.h** file.

The following fields of the *VMount* parameter must be initialized before the call to the **vmount** subroutine:

`vmt_revision`
The revision code in effect when the program that created this virtual file system was compiled. This is the value **VMT_REVISION**.

`vmt_length`
The total length of the structure with all its data. This must be a multiple of the word size (4 bytes) and correspond with the *Size* parameter.

`vmt_flags`
Contains the general mount characteristics. The following value may be specified:

MNT_READONLY
A read-only virtual file system is to be created.

`vmt_gfstype`
The type of the generic file system underlying the **VMT_OBJECT**. Values for this field are defined in the **sys/vmount.h** file and include:

MNT_JFS
Indicates the native file system.

MNT_NFS
Indicates a Network File System client.

MNT_CDROM
Indicates a CD-ROM file system.

vmt_data
An array of structures that describe variable length data associated with the **vmount** structure. The structure consists of the following fields:

vmt_off
The offset of the data from the beginning of the **vmount** structure.

vmt_size
The size, in bytes, of the data.

The array consists of the following fields:

vmt_data[VMT_OBJECT]
Specifies he name of the device, directory, or file to be mounted.

vmt_data[VMT_STUB]
Specifies the name of the device, directory, or file to be mounted over.

vmt_data[VMT_HOST]
Specifies the short (binary) name of the host that owns the mounted object. This need not be specified if **VMT_OBJECT** is local (that is, it has the same `vmt_gfstype` as **/** (root), the root of all file systems).

vmt_data[VMT_HOSTNAME]
Specifies the long (character) name of the host that owns the mounted object. This need not be specified if **VMT_OBJECT** is local.

vmt_data[VMT_INFO]
Specifies binary information to be passed to the generic file-system implementation that supports **VMT_OBJECT**. The interpretation of this field is specific to the `gfs_type`.

vmt_data[VMT_ARGS]
Specifies a character string representation of **VMT_INFO**.

On return from the **vmount** subroutine, the following additional fields of the *VMount* parameter are initialized:

vmt_fsid
Specifies the two-word file system identifier; the interpretation of this identifier depends on the `gfs_type`.

vmt_vfsnumber
Specifies the unique identifier of the virtual file system. Virtual file systems do not survive the IPL; neither does this identifier.

vmt_time
Specifies the time at which the virtual file system was created.

Size
Specifies the size, in bytes, of the supplied data area.

Return Values

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned, and the **errno** global variable is set to indicate the error.

Error Codes

The **mount** and **vmount** subroutines fail and the virtual file system is not created if any of the following is true:

Item	Description
EACCES	The calling process does not have write permission on the stub directory (the directory to be mounted over).

Item	Description
EBUSY	**VMT_OBJECT** specifies a device that is already mounted or an object that is open for writing, or the kernel's mount table is full.
EFAULT	The *VMount* parameter points to a location outside of the allocated address space of the process.
EFBIG	The size of the file system is too big.
EFORMAT	An internal inconsistency has been detected in the file system.
EINVAL	The contents of the *VMount* parameter are unintelligible (for example, the `vmt_gfstype` is unrecognizable, or the file system implementation does not understand the **VMT_INFO** provided).
ENOSYS	The file system type requested has not been configured.
ENOTBLK	The object to be mounted is not a file, directory, or device.
ENOTDIR	The types of **VMT_OBJECT** and **VMT_STUB** are incompatible.
EPERM	**VMT_OBJECT** specifies a block device, and the calling process does not have root user authority.
EROFS	An attempt has been made to mount a file system for read/write when the file system cannot support writing.

vsnprintf Subroutine

Purpose

Print formatted output.

Library

Standard library (**libc.a**)

Syntax

```
#include <stdarg.h>
#include <stdio.h>
```

```
int vsnprintf(char * s, size_t n, const char * format, va_list ap)
```

Description

Refer to **vfprintf**.

vwsprintf Subroutine

Purpose

Writes formatted wide characters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wchar.h>
#include <stdarg.h>
```

```
int vwsprintf (wcs, Format, arg)
wchar_t * wcs;
const char * Format;
va_list  arg;
```

Description

The **vwsprintf** subroutine writes formatted wide characters. It is structured like the **vsprintf** subroutine with a few differences. One difference is that the *wcs* parameter specifies a wide character array into which the generated output is to be written, rather than a character array. The second difference is that the meaning of the **S** conversion specifier is always the same in the case where the **#** flag is specified. If copying takes place between objects that overlap, the behavior is undefined.

Note: The programmer must ensure that there is room for at least `maxlen` wide characters at wcs.

Parameters

Item	Description
wcs	Specifies the array of wide characters where the output is to be written.
Format	Specifies a multibyte character sequence composed of zero or more directives (ordinary multibyte characters and conversion specifiers). The new formats added to handle the wide characters are: **%C** Formats a single wide character. **%S** Formats a wide character string.
arg	Specifies the parameters to be printed.

Return Values

The **vwsprintf** subroutine returns the number of wide characters (not including the terminating wide character null) written into the wide character array and specified by the *wcs* parameter.

w

The following Base Operating System (BOS) runtime services begin with the letter *w*.

wait, waitpid, wait3, wait364, and wait4 Subroutine

Purpose

Waits for a child process to stop or end.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/wait.h>
pid_t wait (StatusLocation)
int *StatusLocation;
pid_t wait ((void *) 0)
```

```
#include <sys/wait.h>
pid_t waitpid (ProcessID,StatusLocation,Options)
int *StatusLocation;
pid_t ProcessID;
int Options;
```

```
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
pid_t wait3 (StatusLocation, Options,ResourceUsage)
int *StatusLocation;
int Options; struct rusage *ResourceUsage;
```

```
pid_t wait364 (StatusLocation, Options,ResourceUsage)
int *StatusLocation;
int Options;
struct rusage64 *ResourceUsage;
```

```
pid_t wait4 (ProcessID,(StatusLocation, Options,ResourceUsage)
pid_t ProcessID;
int *StatusLocation;
int Options;
struct rusage64 *ResourceUsage;
```

Description

The **wait** subroutine suspends the calling thread until the process receives a signal that is not blocked or ignored, or until any one of the calling process' child processes stops or ends. The **wait** subroutine returns without waiting if the child process that is being waited for stops or terminates before the call. On a successful exit, the **pid** of the terminated process is returned by the **wait** subroutine.

Note: The effect of the **wait** subroutine can be modified by the setting of the **SIGCHLD** signal. When **SIGCHLD** is blocked and **wait()** returns because the status of a child process is available and there are no other child processes for which status is available, then any pending **SIGCHLD** signal is cleared. See the **sigaction** (“sigaction, sigvec, or signal Subroutine” on page 1938) subroutine for details.

The **waitpid** subroutine includes a *ProcessID* parameter that allows the calling thread to gather status from a specific set of child processes, according to the following rules:

- If the *ProcessID* value is equal to a value of **-1**, status is requested for any child process. In this respect, the **waitpid** subroutine is equivalent to the **wait** subroutine.
- A *ProcessID* value that is greater than 0 specifies the process ID of a single child process for which status is requested.
- If the *ProcessID* parameter is equal to 0, status is requested for any child process whose process group ID is equal to that of the calling thread's process.
- If the *ProcessID* parameter is less than 0, status is requested for any child process whose process group ID is equal to the absolute value of the *ProcessID* parameter.

The **waitpid**, **wait3**, **wait364**, **wait4** subroutine variants provide an *Options* parameter that can modify the behavior of the subroutine. Two values are defined, **WNOHANG** and **WUNTRACED**, which can be combined by specifying their bitwise-inclusive OR. The **WNOHANG** option prevents the calling thread from being suspended even if there are child processes to wait for. In this case, a value of 0 is returned indicating there are no child processes that stop or terminate. If the **WUNTRACED** option is set, the call also returns information when children of the current process stop because they receive a **SIGTTIN**, **SIGTTOU**, **SIGSSTP**, or **SIGTSTOP** signal.

The **wait364** subroutine can be called to make 64-bit *rusage* counters explicitly available in a 32-bit environment.

The **wait4()** subroutine is similar to the **wait3()** subroutine except that we can specify the process ID of the child. The **wait3()** subroutine waits for any child process but the **wait4()** subroutine can wait for a specific child process.

64-bit quantities are also available to 64-bit applications through the **wait3()** and **wait4()** interface in the *ru_utime* and *ru_stime* fields of **struct rusage**.

When a 32-bit process is being debugged with **ptrace**, the status location is set to **W_SLWTED** if the process calls **load**, **unload**, or **loadbind**. When a 64-bit process is being debugged with **ptrace**, the status location is set to **W_SLWTED** if the process calls **load** or **unload**.

If multiprocessing debugging mode is enabled, the status location is set to **W_SEWTED** if a process is stopped during an exec subroutine and to **W_SFWTED** if the process is stopped during a fork subroutine.

If more than one thread is suspended awaiting termination of the same child process, exactly one thread returns the process status at the time of the child process termination.

If the **WCONTINUED** option is set, the call returns information when the children of the current process continue from a job control stop but whose status is not reported.

Parameters

Item	Description
StatusLocation	Points to integer variable that contains the child process termination status, as defined in the **sys/wait.h** file.
ProcessID	Specifies the child process.
Options	Modifies behavior of subroutine.
ResourceUsage	Specifies the location of a structure to be completed with resource utilization information for terminated children.

Macros

The value pointed to by *StatusLocation* when **wait**, **waitpid**, **wait3**, or **wait4()** subroutines are returned, can be used as the *ReturnedValue* parameter for the following macros that are defined in the **<sys/wait.h>** file to get more information about the process and its child process.

```
WIFCONTINUED(ReturnedValue)
pid_t ReturnedValue;
```

Returns a nonzero value if status returned for a child process that continues from a job control stop.

```
WIFSTOPPED(ReturnedValue)
int ReturnedValue;
```

Returns a nonzero value if status returned for a stopped child.

```
int
WSTOPSIG(ReturnedValue)
int ReturnedValue;
```

Returns the number of the signal that caused the child to stop.

```
WIFEXITED(ReturnedValue)
int ReturnedValue;
```

Returns a nonzero value if status returned for normal termination.

```
int
WEXITSTATUS(ReturnedValue)
int ReturnedValue;
```

Returns the low-order 8 bits of the child exit status.

```
WIFSIGNALED(ReturnedValue)
int ReturnedValue;
```

Returns a nonzero value if status returned for abnormal termination.

```
int
WTERMSIG(ReturnedValue)
int ReturnedValue;
```

Returns the number of the signal that caused the child to terminate.

Return Values

If the **wait** subroutine is unsuccessful, a value of **-1** is returned and the **errno** global variable is set to indicate the error. In addition, the **waitpid**, **wait3**, **wait364**, and **wait4** subroutines return a value of 0 if there are no stopped or exited child processes, and the **WNOHANG** option was specified. The **wait** subroutine returns a 0 if there are no stopped or exited child processes, also.

Error Codes

The **wait**, **waitpid**, **wait3**, **wait364**, and **wait4** subroutines are unsuccessful if one of the following is true:

Item	Description
ECHILD	The calling thread's process has no existing unwaited-for child processes.
EINTR	This subroutine was terminated by receipt of a signal.
EFAULT	The *StatusLocation* or *ResourceUsage* parameter points to a location outside of the address space of the process.

The **waitpid** and **wait4** subroutines are unsuccessful if the following is true:

Item	Description
ECHILD	The process or process group ID specified by the *ProcessID* parameter does not exist or is not a child process of the calling process.

The **waitpid**, **wait3**, and **wait4** subroutines are unsuccessful if the following is true:

Item	Description
EINVAL	The value of the *Options* parameter is not valid.

waitid Subroutine

Purpose

Waits for a child process to change state.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <sys/wait.h>;

int waitid (idtype, id, infop,  options)
idtype_t idtype;
id_t id;
siginfo_t *infop;
int options;
```

Description

The **waitid** subroutine suspends the calling thread until one child of the process containing the calling thread changes state. It records the current state of a child in the structure pointed to by the *infop* parameter. If a child process changed state prior to the call to the **waitid** subroutine, the **waitid** subroutine returns immediately. If more than one thread is suspended in the **wait** or **waitpid** subroutines waiting for termination of the same process, exactly one thread will return the process status at the time of the target process termination.

Parameters

Item	Description
idtype	Specifies the child process.
id	Specifies the child process.
infop	Specifies the location of a **siginfo_t** structure to be filled in with resource utilization information.

Item	Description
options	Specifies which state changes the **waitid** subroutine will wait for. It is formed by OR'ing together one or more of the following flags: **WEXITED** Wait for processes that have exited. **WSTOPPED** Status will be returned for any child that has stopped upon receipt of a signal. **WCONTINUED** Status will be returned for any child that was stopped and has been continued. **WNOHANG** Return immediately if there are no children to wait for. **WNOWAIT** Keep the process whose status is returned in the *infop* parameter in a waitable state. This will not affect the state of the process. The process can be waited for again after this call completes.

Return Values

If **WNOHANG** was specified and there are no children to wait for, 0 is returned. If the **waitid** subroutine returns due to the change of state of one of its children, 0 is returned. Otherwise, -1 is returned and **errno** is set to indicate the error.

Error Codes

The **waitid** subroutine will fail if:

Item	Description
ECHILD	The calling process has no existing unwaited-for child processes.
EINTR	The **waitid** subroutine was interrupted by a signal.
EINVAL	An invalid value was specified for the *options*, or *idtype* parameters and the *id* parameter specifies an invalid set of processes.

wcscat, wcschr, wcscmp, wcscpy, wcpcpy, or wcscspn Subroutine

Purpose

Performs operations on wide-character strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>
```

```
wchar_t * wcscat(WcString1, WcString2)
wchar_t * WcString1;
const wchar_t * WcString2;

wchar_t * wcschr(WcString, WideCharacter)
const wchar_t *WcString;
wchar_t  WideCharacter;

int * wcscmp (WcString1, WcString2)
const wchar_t *WcString1, *WcString2;

wchar_t * wcscpy(WcString1, WcString2)
wchar_t *WcString1;
const wchar_t
*
WcString2;

wchar_t * wcpcpy(WcString1, WcString2)

wchar_t *WcString1;

const wchar_t *WcString2;

size_t  wcscspn(WcString1, WcString2)
const wchar_t *WcString1, *WcString2;
```

Description

The **wcscat**, **wcschr**, **wcscmp**, **wcscpy**, **wcpcpy**, or **wcscspn** subroutine operates on null-terminated **wchar_t** strings. These subroutines expect the string arguments to contain a **wchar_t** null character marking the end of the string. A copy or concatenation operation does not perform boundary checking.

The **wcscat** subroutine copies the contents of the *WcString2* parameter (including the terminating null wide-character code) to the end of the wide-character string pointed to by the *WcString1* parameter. The initial wide-character code of the *WcString2* parameter overwrites the null wide-character code at the end of the *WcString1* parameter. If successful, the **wcscat** subroutine returns the *WcString1* parameter. If the **wcscat** subroutine copies between overlapping objects, the result is undefined.

The **wcschr** subroutine returns a pointer to the first occurrence of the *WideCharacter* parameter in the *WcString* parameter. The character value may be a **wchar_t** null character. The **wchar_t** null character at the end of the string is included in the search. The **wcschr** subroutine returns a pointer to the wide character code, if found, or returns a null pointer if the wide character is not found.

The **wcscmp** subroutine compares two **wchar_t** strings. It returns an integer greater than 0 if the *WcString1* parameter is greater than the *WcString2* parameter. It returns 0 if the two strings are equivalent. It returns a number less than 0 if the *WcString1* parameter is less than the *WcString2* parameter. The sign of the difference in value between the first pair of wide-character codes that differ in the objects being compared determines the sign of a nonzero return value.

The **wcscpy** and the **wcpcpy** subroutines copy the contents of the *WcString2* parameter (including the ending **wchar_t** null character) into the *WcString1* parameter. If successful, the **wcscpy** subroutine returns the *WcString1* parameter and the **wcpcpy** returns a pointer to the terminating null wide-character code copied into the *WcString1*. If these subroutines copy between overlapping objects, the result is undefined.

The **wcscspn** subroutine computes the number of **wchar_t** characters in the initial segment of the string pointed to by the *WcString1* parameter that do not appear in the string pointed to by the *WcString2* parameter. If successful, the **wcscspn** subroutine returns the number of **wchar_t** characters in the segment.

Parameters

Item	Description
WcString1	Points to a wide-character string.
WcString2	Points to a wide-character string.
WideCharacter	Specifies a wide character for location.

Return Values

Upon successful completion, the **wcscat** and **wcscpy** subroutines return a value of ws1. The **wcschr** subroutine returns a pointer to the wide character code. Otherwise, a null pointer is returned.

The **wcpcpy** subroutine returns a pointer to the terminating null wide character code copied into the *ws1*.

The **wcscmp** subroutine returns an integer greater than, equal to, or less than 0, if the wide character string pointed to by the *WcString1* parameter is greater than, equal to, or less than the wide character string pointed to by the *WcString2* parameter.

The **wcscspn** subroutine returns the length of the segment.

wcscoll or wcscoll_l Subroutine

Purpose

Compares wide character strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>

int wcscoll ( WcString1,  WcString2)
const wchar_t *WcString1, *WcString2;

int wcscoll_l ( WcString1, WcString2, Locale)
const wchar_t *WcString1, *WcString2;
locale_t Locale;
```

Description

The **wcscoll** and **wcscoll _l** subroutines compare the two wide-character strings pointed to by the *WcString1* and *WcString2* parameters based on the collation values specified by the **LC_COLLATE** environment variable of the current locale or in the locale represented by *Locale*.

Note: The **wcscoll** subroutine differs from the **wcscmp** subroutine in that the **wcscoll** subroutine compares wide characters based on their collation values, while the **wcscmp** subroutine compares wide characters based on their ordinal values. The **wcscoll** subroutine uses more time than the **wcscmp** subroutine because it obtains the collation values from the current locale.

The **wcscoll** and **wcscoll _l** subroutine may be unsuccessful if the wide character strings specified by the *WcString1* or *WcString2* parameter contains characters outside the domain of the current collating sequence or in the locale represented by the *Locale* collating sequence.

Parameters

Item	Description
WcString1	Points to a wide-character string.
WcString2	Points to a wide-character string.
Locale	Specifies the locale in which character has to be converted.

Return Values

The **wcscoll** and **wcscoll _l** subroutine returns the following values:

Item	Description
< 0	The collation value of the *WcString1* parameter is less than that of the *WcString2* parameter.
=0	The collation value of the *WcString1* parameter is equal to that of the *WcString2* parameter.
>0	The collation value of the *WcString1* parameter is greater than that of the *WcString2* parameter.

The **wcscoll** and **wcscoll _l** subroutines indicate error conditions by setting the **errno** global variable. However, there is no return value to indicate an error. To check for errors, the **errno** global variable should be set to 0, then checked upon return from the **wcscoll**, and **wcscoll _l** subroutines. If the **errno** global variable is nonzero, an error occurred.

Error Codes

Item	Description
EINVAL	The *WcString1* or *WcString2* arguments contain wide-character codes outside the domain of the collating sequence.

wcsftime Subroutine

Purpose

Converts date and time into a wide character string.

Library

Standard C Library (**libc. a**)

Syntax

```
#include <time.h>

size_t wcsftime (WcString, Maxsize, Format, TimPtr)
wchar_t * WcString;
size_t  Maxsize;
const wchar_t * Format;
const struct tm * TimPtr;
```

Description

The **wcsftime** function is equivalent to the **strftime** function, except that:

- The argument *wcs* points to the initial element of an array of wide-characters into which the generated output is to be placed.
- The argument *maxsize* indicates the maximum number of wide-characters to be placed in the output array.
- The argument *format* is a wide-character string and the conversion specifications are replaced by corresponding sequences of wide-characters.
- The return value indicates the number of wide-characters placed in the output array.

If copying takes place between objects that overlap, the behavior is undefined.

Parameters

Item	Description
WcString	Contains the output of the **wcsftime** subroutine.
Maxsize	Specifies the maximum number of bytes (including the wide character null-terminating byte) that may be placed in the *WcString* parameter.
Format	Specifiers are the same as in **strftime** (“strftime or strftime_l Subroutine” on page 2075) function.
TimPtr	Contains the data to be converted by the **wcsftime** subroutine.

Return Values

If successful, and if the number of resulting wide characters (including the wide character null-terminating byte) is no more than the number of bytes specified by the *Maxsize* parameter, the **wcsftime** subroutine returns the number of wide characters (not including the wide character null-terminating byte) placed in the *WcString* parameter. Otherwise, 0 is returned and the contents of the *WcString* parameter are indeterminate.

wcsid Subroutine

Purpose

Returns the charsetID of a wide character.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

int wcsid ( WC)
const wchar_t WC;
```

Description

The **wcsid** subroutine returns the charsetID of the **wchar_t** character. No validation of the character is performed. The parameter must point to a value in the character range of the current code set defined in the current locale.

Parameters

Item	Description
WC	Specifies the character to be tested.

Return Values

Successful completion returns an integer value representing the charsetID of the character. This integer can be a number from 0 through *n*, where *n* is the maximum character set defined in the CHARSETID field of the **charmap**.

wcslen, or wcsnlen Subroutine

Purpose

Determines the number of characters in a wide-character string.

Library

Standard C Library (**libc.a**)

Syntax

#include <wcstr.h>

size_t wcslen(*WcString***) const wchar_t ****WcString***;**

size_t wcsnlen(*WcString*, *maxlen***)**

const wchar_t ****WcString*;**

size_t *maxlen*

Description

The **wcslen** subroutine computes the number of **wchar_t** characters in the string pointed to by the *WcString* parameter.

The **wcsnlen** subroutine computes the smaller of the number of wide characters in the string pointed by *WcString*, not including the terminating null wide character code, and the value of *maxlen*. The **wcsnlen** subroutine does not examine more than the first *maxlen* characters of the wide character string pointed to by *WcString*.

Parameters

Item	Description
WcString	Specifies a wide-character string.

Return Values

The **wcslen** subroutine returns the number of **wchar_t** characters that precede the terminating **wchar_t** null character.

The **wcsnlen** subroutine returns an integer containing the smaller of either the length of the wide character string pointed to by *WcString* or *maxlen*.

wcsncat, wcsncmp, wcsncpy, or wcpncpy Subroutine

Purpose

Performs operations on a specified number of wide characters from one string to another.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wcstr.h>
```

wchar_t * wcsncat (*WcString1*, *WcString2*, *Number*)
wchar_t * *WcString1*;
const wchar_t * *WcString2*;
size_t *Number*;

wchar_t * wcsncmp (*WcString1*, *WcString2*, *Number*)
**const wchar_t ***_WcString1_, ***_WcString2_;
size_t *Number*;

wchar_t * wcsncpy (*WcString1*, *WcString2*, *Number*)
**wchar_t ***_WcString1_;
**const wchar_t ***_WcString2_;
size_t *Number*;

wchar_t * wcpncpy (*WcString1*, *WcString2*, *Number***)**
**wchar_t ***_WcString1_;
**const wchar_t ***_WcString2_;
size_t *Number*;

Description

The **wcsncat**, **wcsncmp**, **wcsncpy**, and **wcpncpy** subroutines operate on null-terminated wide character strings.

The **wcsncat** subroutine appends characters from the *WcString2* parameter, up to the value of the *Number* parameter, to the end of the *WcString1* parameter. It appends a **wchar_t** null character to the result and returns the *WcString1* value.

The **wcsncmp** subroutine compares wide characters in the *WcString1* parameter, up to the value of the *Number* parameter, to the *WcString2* parameter. It returns an integer greater than 0 if the value of the *WcString1* parameter is greater than the value of the *WcString2* parameter. It returns a 0 if the strings are equivalent. It returns an integer less than 0 if the value of the *WcString1* parameter is less than the value of the *WcString2* parameter.

The **wcsncpy**, and **wcpncpy** subroutines copies wide characters from the *WcString2* parameter, up to the value of the *Number* parameter, to the *WcString1* parameter. It returns the value of the *WcString1* parameter. If the number of characters in the *WcString2* parameter is less than the *Number* parameter, the *WcString1* parameter is padded out with **wchar_t null** characters to a number equal to the value of the *Number* parameter.

If any null wide character codes is written into the destination, the **wcpncpy** subroutine returns the address of the first such null wide character code. Otherwise, it returns *& WcString1 [Number]*.

Parameters

Item	Description
WcString1	Specifies a wide-character string.
WcString2	Specifies a wide-character string.
Number	Specifies the range of characters to process.

wcspbrk Subroutine

Purpose

Locates the first occurrence of characters in a string.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>
```

```
wchar_t *wcspbrk( WcString1,  WcString2)
const wchar_t *WcString1;
const wchar_t *WcString2;
```

Description

The **wcspbrk** subroutine locates the first occurrence in the wide character string pointed to by the *WcString1* parameter of any wide character from the string pointed to by the *WcString2* parameter.

Parameters

Item	Description
WcString1	Points to a wide-character string being searched.
WcString2	Points to a wide-character string.

Return Values

If no **wchar_t** character from the *WcString2* parameter occurs in the *WcString1* parameter, the **wcspbrk** subroutine returns a pointer to the wide character, or a null value.

wcsrchr Subroutine

Purpose

Locates a **wchar_t** character in a wide-character string.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wcstr.h>
```

```
wchar_t *wcsrchr ( WcString,  WideCharacter)
const wchar_t *WcString;
wint_t WideCharacter;
```

Description

The **wcsrchr** subroutine locates the last occurrence of the *WideCharacter* value in the string pointed to by the *WcString* parameter. The terminating **wchar_t** null character is considered to be part of the string.

Parameters

Item	Description
WcString	Points to a string.
WideCharacter	Specifies a **wchar_t** character.

Return Values

The **wcsrchr** subroutine returns a pointer to the *WideCharacter* parameter value, or a null pointer if that value does not occur in the specified string.

wcsrtombs, or wcsnrtombs Subroutine

Purpose

Convert a wide-character string to a character string (restartable).

Library

Standard library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
size_t wcsrtombs (char * dst, const wchar_t ** src, size_t len, mbstate_t * ps);
size_t wcsnrtombs (char * dst, const wchar_t ** src, size_t nwc, size_t len, mbstate_t * ps);
```

Description

The **wcsrtombs** function converts a sequence of wide-characters from the array indirectly pointed to by **src** into a sequence of corresponding characters, beginning in the conversion state described by the object pointed to by **ps**.If **dst** is not a null pointer, the converted characters are then stored into the array pointed to by **dst**. Conversion continues up to and including a terminating null wide-character, which is also stored. Conversion stops earlier in the following cases:

- When a code is reached that does not correspond to a valid character.
- When the next character would exceed the limit of **len** total bytes to be stored in the array pointed to by **dst** (and **dst** is not a null pointer).

Each conversion takes place as if by a call to the **wcrtomb** function.

If **dst** is not a null pointer, the pointer object pointed to by **src** is assigned either a null pointer (if conversion stopped due to reaching a terminating null wide-character) or the address just past the last wide-character converted (if any). If conversion stopped due to reaching a terminating null wide-character, the resulting state described is the initial conversion state.

If ps is a null pointer, the **wcsrtombs** function uses its own internal **mbstate_t** object, which is initialised at program startup to the initial conversion state. Otherwise, the **mbstate_t** object pointed to by **ps** is used to completely describe the current conversion state of the associated character sequence. The implementation will behave as if no function defined in this specification calls **wcsrtombs**.

The **wcsnrtombs** function is equivalent to the **wcsrtombs** function, except that the conversion is limited to the first *nwc* wide characters.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

If conversion stops because a code is reached that does not correspond to a valid character, an encoding error occurs. In this case, the **wcsrtombs** and **wcsnrtombs** functions store the value of the macro EILSEQ in **errno** and returns **(size_t)-1**; the conversion state is undefined. Otherwise, the **wcsrtombs** and **wcsnrtombs** functions return the number of bytes in the resulting character sequence, not including the terminating null (if any).

Error Codes

The **wcsrtombs** function may fail if:

Item	Description
EINVAL	**ps** points to an object that contains an invalid conversion state.
EILSEQ	A wide-character code does not correspond to a valid character.

wcsspn Subroutine

Purpose

Returns the number of wide characters in the initial segment of a string.

Library

Standard C Library (**libc.a**)

Syntax

#include <wcstr.h>

size_t wcsspn(*WcString1*, *WcString2***) const wchar_t *****WcString1*, *****WcString2*;

Description

The **wcsspn** subroutine computes the number of **wchar_t** characters in the initial segment of the string pointed to by the *WcString1* parameter. The *WcString1* parameter consists entirely of **wchar_t** characters from the string pointed to by the *WcString2* parameter.

Parameters

Item	Description
WcString1	Points to the initial segment of a string.

Item	Description
WcString2	Points to a set of characters string.

Return Values

The **wcsspn** subroutine returns the number of **wchar_t** characters in the segment.

wcsstr Subroutine

Purpose

Find a wide-character substring.

Library

Standard library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
wchar_t *wcsstr (const wchar_t * ws1, const wchar_t * ws2);
```

Description

The **wcsstr** function locates the first occurrence in the wide-character string pointed to by **ws1** of the sequence of wide-characters (excluding the terminating null wide-character) in the wide- character string pointed to by **ws2**.

Return Values

On successful completion, **wcsstr** returns a pointer to the located wide-character string, or a null pointer if the wide-character string is not found.

If **ws2** points to a wide-character string with zero length, the function returns **ws1**.

wcstod, wcstof, or wcstold Subroutine

Purpose

Converts a wide character string to a double-precision number.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
#include <wchar.h>
```

```
double wcstod ( nptr,  endptr)
const wchar_t *nptr;
wchar_t **endptr;
```

```
float wcstof (nptr, endptr)
const wchar_t *restrict  nptr;
wchar_t **restrict endptr;

long double wcstold (nptr, endptr)
const wchar_t *restrict format;
wchar_t **restrict nptr;
```

Description

The **wcstod**, **wcstof**, and **wcstold** subroutines convert the initial portion of the wide-character string pointed to by *nptr* to **double**, **float** and **long double** representation, respectively. First, they decompose the input wide-character string into three parts:

- An initial, possibly empty, sequence of white-space wide-character codes.
- A subject sequence interpreted as a floating-point constant or representing infinity or NaN.
- A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string.

Then they convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, and one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, and an optional exponent part.
- A 0x or 0X, and a non-empty sequence of hexadecimal digits optionally containing a radix character, and an optional binary exponent part.
- One of INF or INFINITY, or any other wide string equivalent except for case.
- One of NAN or NAN(*n-wchar-sequence* $_{opt}$), or any other wide string ignoring case in the NAN part, where:

```
n-wchar-sequence:
         digit
         nondigit
         n-wchar-sequence digit
         n-wchar-sequence nondigit
```

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the radix character (whichever occurs first) are interpreted as a floating constant according to the rules of the C language, except that the radix character is used in place of a period. If neither an exponent part or a radix character appears in a decimal floating-point number, or if a binary exponent part does not appear in a hexadecimal floating-point number, an exponent part of the appropriate type with value zero is assumed to follow the last digit in the string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide-character sequence INF or INFINITY is interpreted as an infinity, if representable in the return type, or else as if it were a floating constant that is too large for the range of the return type. A wide-character sequence NAN or NAN(*n-wchar-sequence* $_{opt}$) is interpreted as a quiet NaN, if supported in the return type, or else as if it were a subject sequence part that does not have the expected form. The meaning of the *n*-wchar sequences is implementation-defined. A pointer to the final wide string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If the subject sequence has the hexadecimal form and FLT_RADIX is a power of 2, the conversion will be rounded in an implementation-defined manner.

The radix character is as defined in the program's locale (category *LC_NUMERIC*). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period.

In other than the C or POSIX locales, other implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The **wcstod**, **wcstof**, and **wcstold** subroutines do not change the setting of the **errno** global variable if successful.

Since 0 is returned on error and is also a valid return on success, an application wishing to check for error situations should set **errno** to 0, call **wcstod**, **wcstof**, or **wcstold**, and check **errno**.

Parameters

Item	Description
nptr	Contains a pointer to the wide character string to be converted to a double-precision value.
endptr	Contains a pointer to the position in the string specified by the *nptr* parameter where a wide character is found that is not a valid character for the purpose of this conversion.

Return Values

Upon successful completion, the **wcstod**, **wcstof**, and **wcstold** subroutines return the converted value. If no conversion could be performed, 0 is returned and the **errno** global variable may be set to **EINVAL**.

If the correct value is outside the range of representable values, plus or minus **HUGE_VAL**, **HUGE_VALF**, or **HUGE_VALL** is returned (according to the sign of the value), and **errno** is set to ERANGE.

If the correct value would cause underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type is returned and **errno** set to ERANGE.

wcstod32, wcstod64, or wcstod128 Subroutine

Purpose

Converts a wide character string to a decimal floating-point number.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
#include <wchar.h>

_Decimal32 wcstod32 (nptr, endptr)
const wchar_t *nptr;
wchar_t **endptr;

_Decimal64 wcstod64 (nptr, endptr)
const wchar_t *nptr;
wchar_t **endptr;

_Decimal128 wcstod128 (nptr, endptr)
const wchar_t *nptr;
wchar_t **endptr;
```

Description

The **wcstod32**, **wcstod64**, and **wcstod128** subroutines convert the initial portion of the wide-character string pointed to by the *nptr* parameter to **_Decimal32**, **_Decimal64**, and **_Decimal128** representation, respectively. First, these subroutines decompose the input wide-character string into three parts:

- An initial and possibly empty sequence of white-space and wide-character codes
- A subject sequence interpreted as a floating-point constant or represents infinity or NaN
- A final wide-character string of one or more unrecognized wide-character codes, including the terminating null wide-character code of the input wide-character string

Then, **wcstod32**, **wcstod64**, and **wcstod128** subroutines attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign and one of the following:

- A non-empty sequence of decimal digits that might contains a radix character and an exponent part
- INF, INFINITY, or any other wide string equivalent except for case
- NAN or NAN (*n-wchar-sequence* $_{opt}$), ignoring case in the NAN, where:

```
n-wchar-sequence:
               digit
               n-wchar-sequence digit
```

The subject sequence is defined as the longest initial subsequence of the input wide string, starting with the first non-white-space wide character that is of the expected form. The subject sequence contains no wide characters if the input wide string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the radix character (whichever occurs first) are interpreted as a floating constant according to the rules of the C language, except that the sequence is not a hexadecimal floating number, or that the radix character is used in place of a period. If neither an exponent part nor a radix character appears in a decimal floating-point number, an exponent part of the appropriate type with a value of 0 is assumed to follow the last digit in the string.

If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide-character sequence INF or INFINITY is interpreted as infinity. A wide-character sequence NAN or NAN(*n-wchar-sequence* $_{opt}$) is interpreted as a quiet NaN. The meaning of the *n*-wchar sequences is implementation-defined. A pointer to the final wide string is stored in the object pointed to by the *endptr* parameter, provided that the *endptr* parameter is not a null pointer.

The radix character is as defined in the locale of the program (category LC_NUMERIC). In the POSIX locale, or in a locale where the radix character is not defined, the radix character defaults to a period.

In locales other than the C or POSIX locale, other implementation-defined subject sequences can be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed. The value of the *nptr* parameter is stored in the object pointed to by the *endptr* parameter, provided that the *endptr* parameter is not a null pointer.

The **wcstod32**, **wcstod64**, and **wcstod128** subroutines do not change the setting of the **errno** global variable if successful.

A value of 0 is returned on error and is also a valid return on success. Therefore, an application wishing to check for error situations should set the **errno** global variable to the value of 0, call the **wcstod32**, **wcstod64**, or **wcstod128** subroutine, and check the **errno** global variable.

Parameters

Item	Description
nptr	Contains a pointer to the string to be converted to a decimal floating point value.
endpr	Contains a pointer to the position in the string specified by the *nptr* parameter where a wide character is found that is not a valid character for the conversion.

Return Values

Upon successful completion, the **wcstod32**, **wcstod64**, and **wcstod128** subroutines return the converted value. If no conversion can be performed, the value of 0 is returned and the **errno** global variable might be set to EINVAL.

If the correct value is outside the range of representable values, **±HUGE_VAL_D32**, **±HUGE_VAL_D64**, or **±HUGE_VAL_D128** is returned (according to the return type and sign of the value), and the **errno** global variable is set to ERANGE.

If the correct value causes underflow, a value whose magnitude is no greater than the smallest normalized positive number in the return type is returned, and the **errno** global variable is set to ERANGE.

wcstoimax or wcstoumax Subroutine

The **wcstoimax** or **wcstoumax** subroutines are equivalent to the **wcstol**, **wcstoll**, **wcstoul**, and **wcstoull** subroutines, respectively, except that the initial portion of the wide string is converted to **intmax_t** and **uintmax_t** representation, respectively.

Purpose

Converts a wide-character string to an integer type.

Syntax

```
#include <stddef.h>
#include <inttypes.h>

intmax_t wcstoimax (nptr, endptr, base)
const wchar_t *restrict nptr;
wchar_t **restrict endptr;
int base;

uintmax_t wcstoumax (nptr, endptr, base)
const wchar_t *restrict  nptr;
wchar_t **restrict endptr;
int base;
```

Description

Parameters

Item	Description
nptr	Points to the wide-character string.
endptr	Points to the object where the final wide-character string is stored.
base	Determines the subject sequence interpreted as an integer.

Return Values

The **wcstoimax** or **wcstoumax** subroutines return the converted value, if any.

If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, {**INTMAX_MAX**}, {**INTMAX_MIN**}, or {**UINTMAX_MAX**} is returned (according to the return type and sign of the value, if any), and the **errno** global variable is set to ERANGE.

wcstok Subroutine

Purpose

Converts wide-character strings to tokens.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wchar.h>

wchar_t *wcstok ( WcString1,  WcString2,  ptr)
wchar_t *WcString1;
const wchar_t *WcString2;
wchar_t **ptr
```

Description

A sequence of calls to the **wcstok** subroutine breaks the wide-character string pointed to by *WcString1* into a sequence of tokens, each of which is delimited by a wide-character code from the wide-character string pointed to by *WcString2*. The third argument points to a caller-provided **wchar_t** pointer where **wcstok** stores information necessary for it to continue scanning the same wide-character string.

The first call in the sequence has *WcString1* as its first argument and is followed by calls with a nullpointer as their first argument. The separator string pointed to by *WcString2* may be different from call to call.

The first call in the sequence searches the wide-character string pointed to by *WcString1* for the first wide-character code that is not contained in the current separator string pointed to by *WcString2*. If no such wide-character code is found, then there are no tokens in the wide-character string pointed to by *WcString1* and **wcstok** returns a null pointer. If such a wide-character code is found, it is the start of the first token.

The **wcstok** subroutine then searches from there for a wide-character code that is contained in the current separator string. If no such wide-character code is found, the current token extends to the end of the wide-character string pointed to by *WcString1*, and subsequent searches for a token returns a null pointer. If such a wide-character code is found, it is overwritten by a null wide-character, which terminates the current token. The **wcstok** subroutine saves a pointer to the following wide-character code, from which the next search for a token starts.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation behaves as if no function calls **wcstok**.

Parameters

Item	Description
ptr	Contains a pointer to a caller-provided **wchar_t** pointer where **wcstok** stores information necessary for it to continue scanning the same wide-character string.
WcString1	Contains a pointer to the wide-character string to be searched.
WcString2	Contains a pointer to the string of wide-character token delimiters.

Return Values

Upon successful completion, **wcstok** returns a pointer to the first wide-character code of a token. Otherwise, if there is no token, **wcstok** returns a null pointer.

Examples

To convert a wide-character string to tokens, use the following:

```
#include <wchar.h>
#include <locale.h>
#include <stdlib.h>
```

```
main()
{
        wchar_t WCString1[] = L"?a???b,,,#c";
        wchar_t *ptr;
        wchar_t *pwcs;
```

```
        (void)setlocale(LC_ALL, "");
        pwcs = wcstok(WCString1, L"?", &ptr);
                /* pwcs points to the token L"a"*/
        pwcs = wcstok((wchar_t *)NULL, L",", &ptr);
                /* pwcs points to the token L"??b"*/
        pwcs = wcstok( (wchar_t *)NULL, L"#,", &ptr);
                /* pwcs points to the token L"c"*/

}
```

wcstol or wcstoll Subroutine

Purpose

Converts a wide-character string to a long integer representation.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

long int wcstol (*Nptr*, *Endptr*, *Base*)
const wchar_t **Nptr*;
wchar_t ***Endptr*;
int *Base*;

```
long long int wcstoll (*Nptr, **Endptr, Base)
const wchar_t *Nptr;
wchar_t **Endptr:
int Base
```

Description

The **wcstol** subroutine converts a wide-character string to a long integer representation. The **wcstoll** subroutine converts a wide-character string to a long long integer representation.

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by the **iswspace** subroutine)
2. A subject sequence interpreted as an integer and represented in a radix determined by the *Base* parameter
3. A final wide-character string of one or more unrecognized wide-character codes, including the terminating wide-character null of the input wide-character string

If possible, the subject is then converted to an integer, and the result is returned.

The *Base* parameter can take the following values: 0 through 9, or a (or A) through z (or Z). There are potentially 36 values for the base. If the base value is 0, the expected form of the subject string is that of a decimal, octal, or hexadecimal constant, any of which can be preceded by a + (plus sign) or - (minus sign). A decimal constant starts with a non zero digit, and is composed of a sequence of decimal digits. An octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7. A hexadecimal constant is defined as the prefix 0x (or 0X) followed by a sequence of decimal digits and the letters a (or A) to f (or F) with values ranging from 10 (for a or A) to 15 (for f or F).

If the base value is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer in the radix specified by the *Base* parameter, optionally preceded by a + or -, but not including an integer suffix. The letters a (or A) through z (or Z) are ascribed the values of 10 to 35. Only letters whose values are less than that of the base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters or digits, following the sign, if present.

The wide-character string is parsed to skip the initial space characters (as determined by the **iswspace** subroutine). Any non-space character signifies the start of a subject string that may form an integer in the radix specified by the *Base* parameter. The subject sequence is defined to be the longest initial substring that is a long integer of the expected form. Any character not satisfying this form begins the final portion of the wide-character string pointed to by the *Endptr* parameter on return from the call to the **wcstol** or **wcstoll** subroutine.

Parameters

Item	Description
Nptr	Contains a pointer to the wide-character string to be converted to a long integer number.
Endptr	Contains a pointer to the position in the *Nptr* parameter string where a wide-character is found that is not a valid character.
Base	Specifies the radix in which the characters are interpreted.

Return Values

The **wcstol** and **wcstoll** subroutines return the converted value of the long or long long integer if the expected form is found. If no conversion could be performed, a value of 0 is returned. If the converted value is outside the range of representable values, **LONG_MAX** or **LONG_MIN** is returned for the **wcstol** subroutine and **LLONG_MAX** or **LLONG_MIN** is returned for the **wcstoll** subroutine (according to the sign of the value). The value of **errno** is set to **ERANGE**. If the base value specified by the *Base* parameter is not supported, **EINVAL** is returned.

If the subject sequence has the expected form, it is interpreted as an integer constant in the appropriate base. A pointer to the final string is stored in the *Endptr* parameter if that parameter is not a null pointer.

If the subject sequence is empty or does not have a valid form, no conversion is done. The value of the *Nptr* parameter is stored in the *Endptr* parameter if that parameter is not a null pointer.

Since 0, **LONG_MIN**, and **LONG_MAX** (for **wcstol**) and **LLONG_MIN**, and **LLONG_MAX** (for **wcstoll**) are returned in the event of an error and are also valid returns if the **wcstol** or **wcstoll** subroutine is successful, applications should set the **errno** global variable to 0 before calling either subroutine, and check **errno** after return. If the **errno** global value has changed, an error occurred.

Examples

To convert a wide-character string to a signed long integer, use the following code:

```
#include <stdlib.h>
#include <locale.h>
#include <errno.h>
```

```
main()
{
        wchar_t *WCString, *endptr;
        long int retval;
        (void)setlocale(LC_ALL, "");
        /**Set errno to 0 so a failure for wcstol can be
        **detected */
        errno=0;
        /*
        **Let WCString point to a wide character null terminated
        ** string containing a signed long integer value
        **
                */retval = wcstol ( WCString &endptr, 0 );
        /* Check errno, if it is non-zero, wcstol failed */
        if (errno != 0) {
                /*Error handling*/
        }
        else if (&WCString == endptr) {
                /* No conversion could be performed */
                /* Handle this case accordingly. */
        }
        /* retval contains long integer */
}
```

wcstombs Subroutine

Purpose

Converts a sequence of wide characters into a sequence of multibyte characters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

size_t wcstombs (*String*, *WcString*, *Number*)
char ***String**;
const wchar_t ***WcString**;
size_t *Number*;

Description

The **wcstombs** subroutine converts the sequence of wide characters pointed to by the *WcString* parameter to a sequence of corresponding multibyte characters and places the results in the area pointed to by the *String* parameter. The conversion is terminated when the null wide character is encountered or when the number of bytes specified by the *Number* parameter (or the value of the *Number* parameter minus 1) has been placed in the area pointed to by the *String* parameter. If the amount of space available in the area pointed to by the *String* parameter would cause a partial multibyte character to be stored, the subroutine uses a number of bytes equalling the value of the *Number* parameter minus 1, because only complete multibyte characters are allowed.

Parameters

Item	Description
String	Points to the area where the result of the conversion is stored. If the *String* parameter is a null pointer, the subroutine returns the number of bytes required to hold the conversion.
WcString	Points to a wide-character string.
Number	Specifies a number of bytes to be converted.

Return Values

The **wcstombs** subroutine returns the number of bytes modified. If a wide character is encountered that is not valid, a value of -1 is returned.

Error Codes

The **wcstombs** subroutine is unsuccessful if the following error occurs:

Item	Description
EILSEQ	An invalid character sequence is detected, or a wide-character code does not correspond to a valid character.

wcstoul or wcstoull Subroutine

Purpose

Converts wide character strings to unsigned long or long long integer representation.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>

unsigned long int wcstoul (Nptr, Endptr, Base)
const wchar_t * Nptr;
wchar_t ** Endptr;
int  Base;

unsigned long long int wcstoull (Nptr, Endptr, Base)
const wchar_t *Nptr;
wchar_t **Endptr;
int Base;
```

Description

The **wcstoul** and **wcstoull** subroutines convert the initial portion of the wide character string pointed to by the *Nptr* parameter to an unsigned long or long long integer representation. To do this, it parses the wide character string pointed to by the *Nptr* parameter to obtain a valid string (that is, subject string) for the purpose of conversion to an unsigned long integer. It then points the *Endptr* parameter to the position where an unrecognized character, including the terminating null, is found.

The base specified by the *Base* parameter can take the following values: 0 through 9, a (or A) through z (or Z). There are potentially 36 values for the base. If the base value is 0, the expected form of the subject string is that of an unsigned integer constant, with an optional + (plus sign) or - (minus sign), but

not including the integer suffix. If the base value is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by the *Base* parameter, optionally preceded by a + or -, but not including an integer suffix.

The letters a (or A) through z (or Z) are ascribed the values of 10 to 35. Only letters whose values are less than that of the base are permitted. If the value of the base is 16, the characters 0x (or 0X) may optionally precede the sequence of letters or digits, following a + or - . present.

The wide character string is parsed to skip the initial white-space characters (as determined by the **iswspace** subroutine). Any nonspace character signifies the start of a subject string that may form an unsigned long integer in the radix specified by the *Base* parameter. The subject sequence is defined to be the longest initial substring that is an unsigned long integer of the expected form. Any character not satisfying this expected form begins the final portion of the wide character string pointed to by the *Endptr* parameter on return from the call to this subroutine.

Parameters

Item	Description
Nptr	Contains a pointer to the wide character string to be converted to an unsigned long integer.
Endptr	Contains a pointer to the position in the *Nptr* string where a wide character is found that is not a valid character for the purpose of this conversion.
Base	Specifies the radix in which the wide characters are interpreted.

Return Values

The **wcstoul** and **wcstoull** subroutines return the converted value of the unsigned long or long long integer if the expected form is found. If no conversion could be performed, a value of 0 is returned. If the converted value is outside the range of representable values, a **ULONG_MAX** value is returned (for **wcstoul**), and **ULLONG_MAX** is returned (for **wcstoull**), and the value of the **errno** global variable is set to a **ERANGE** value.

If the subject sequence has the expected form, it is interpreted as an integer constant in the appropriate base. A pointer to the final string is stored in the *Endptr* parameter if that parameter is not a null pointer. If the subject sequence is empty or does not have a valid form, no conversion is done and the value of the *Nptr* parameter is stored in the *Endptr* parameter if it is not a null pointer.

If the radix specified by the *Base* parameter is not supported, an **EINVAL** value is returned. If the value to be returned is not representable, an **ERANGE** value is returned.

Examples

To convert a wide character string to an unsigned long integer, use the following code:

```
#include <stdlib.h>
#include <locale.h>
#include <errno.h>
extern int errno;
```

```
main()
{
        wchar_t *WCString, *EndPtr;
        unsigned long int   retval;
```

```
        (void)setlocale(LC_ALL, "");
        /*
        ** Let WCString point to a wide character null terminated
        ** string containing an unsigned long integer value.
        **
        */
```

```
        retval = wcstoul ( WCString  &EndPtr,  0 );
        if(retval==0) {
                /* No conversion could be performed */
                /* Handle this case accordingly. */
        } else if(retval == ULONG_MAX) {
                /* Error handling */
        }
        /* retval contains the unsigned long integer value. */
}
```

wcswcs Subroutine

Purpose

Locates first occurrence of a wide character in a string.

Library

Standard C Library (**libc.a**)

Syntax

#include <string.h>

wchar_t *wcswcs(*WcString1*, *WcString2***) const wchar_t ***WcString1***, ***WcString2***;**

Description

The **wcswcs** subroutine locates the first occurrence, in the string pointed to by the *WcString1* parameter, of a sequence of **wchar_t** characters (excluding the terminating **wchar_t** null character) from the string pointed to by the *WcString2* parameter.

Parameters

Item	Description
WcString1	Points to the wide-character string being searched.
WcString2	Points to a wide-character string, which is a source string.

Return Values

The **wcswcs** subroutine returns a pointer to the located string, or a null value if the string is not found. If the *WcString2* parameter points to a string with 0 length, the function returns the *WcString1* value.

wcswidth Subroutine

Purpose

Determines the display width of wide character strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>
```

```
int wcswidth (* Pwcs,  n)
const wchar_t *Pwcs;
size_t n;
```

Description

The **wcswidth** subroutine determines the number of display columns to be occupied by the number of wide characters specified by the *N* parameter in the string pointed to by the *Pwcs* parameter. The **LC_CTYPE** category affects the behavior of the **wcswidth** subroutine. Fewer than the number of wide characters specified by the *N* parameter are counted if a null character is encountered first.

Parameters

Item	Description
N	Specifies the maximum number of wide characters whose display width is to be determined.
Pwcs	Contains a pointer to the wide character string.

Return Values

The **wcswidth** subroutine returns the number of display columns to be occupied by the number of wide characters (up to the terminating wide character null) specified by the *N* parameter (or fewer) in the string pointed to by the *Pwcs* parameter. A value of zero is returned if the *Pwcs* parameter is a wide character null pointer or a pointer to a wide character null (that is, *Pwcs* or ***Pwcs* is null). If the *Pwcs* parameter points to an unusable wide character code, -1 is returned.

Examples

To find the display column width of a wide character string, use the following:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>
```

```
main()
{
   wchar_t *pwcs;
   int     retval, n ;
```

```
   (void)setlocale(LC_ALL, "");
   /* Let pwcs point to a wide character null terminated
   ** string. Let n be the number of wide characters whose
   ** display column width is to be determined.
   */
   retval=  wcswidth( pwcs, n );
   if(retval == -1){
           /* Error handling. Invalid wide character code
           ** encountered in the wide character string pwcs.
           */
   }
}
```

wcsxfrm Subroutine

Purpose

Transforms wide-character strings to wide-character codes of current locale.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>
```

size_t wcsxfrm (*WcString1*, *WcString2*, *Number*)
wchar_t *WcString1;
const wchar_t *WcString2;
size_t *Number*;

size_t wcsxfrm_l (*WcString1*,
WcString2, *Number*, *Locale*)
wchar_t* *WcString1*;
const wchar_t* *WcString2*;
size_t *Number*;
locale_t *Locale*;

Description

The **wcsxfrm** and **wcsxfrm _l** subroutines transform the wide-character string specified by the *WcString2* parameter into a string of wide-character codes, based on the collation values of the wide characters in the current locale as specified by the **LC_COLLATE** category of the current locale or the locale represented by *Locale* respectively. No more than the number of character codes specified by the *Number* parameter are copied into the array specified by the *WcString1* parameter. When two such transformed wide-character strings are compared using the **wcscmp** or **wcscoll_l** subroutine, the result is the same as that obtained by a direct call to the **wcscoll** or **wcscoll_l** the subroutine on the two original wide-character strings.

Parameters

Item	Description
WcString1	Points to the destination wide-character string.
WcString2	Points to the source wide-character string.
Number	Specifies the maximum number of wide-character codes to place into the array specified by *WcString1*. To determine the necessary size specification, set the *Number* parameter to a value of 0, so that the *WcString1* parameter becomes a null pointer. The return value plus 1 is the size necessary for the conversion.
Locale	Specifies the locale in which character has to be converted.

Return Values

If the *WcString1* parameter is a wide-character null pointer, the **wcsxfrm** and the **wcsxfrm_l** subroutine return the number of wide-character elements (not including the wide-character null terminator) required to store the transformed wide character string. If the count specified by the *Number* parameter is sufficient to hold the transformed string in the *WcString1* parameter, including the wide character null terminator, the return value is set to the actual number of wide character elements placed in the *WcString1* parameter, not including the wide character null. If the return value is equal to or greater than the value specified by the *Number* parameter, the contents of the array pointed to by the *WcString1* parameter are indeterminate. This occurs whenever the *Number* value parameter is too small to hold the entire transformed string. If an error occurs, the **wcsxfrm** subroutine returns the **size_t** data type with a value of -1 and sets the **errno** global variable to indicate the error.

If the wide character string pointed to by the *WcString2* parameter contains wide character codes outside the domain of the collating sequence defined by the current locale, the **wcsxfrm** and **wcsxfrm_l** subroutines return a value of **EINVAL**.

wctob Subroutine

Purpose

Wide-character to single-byte conversion.

Library

Standard library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <wchar.h>
```

```
int wctob (wint_t c);
```

Description

The **wctob** function determines whether **c** corresponds to a member of the extended character set whose character representation is a single byte when in the initial shift state.

The behavior of this function is affected by the LC_CTYPE category of the current locale.

Return Values

The **wctob** function returns EOF if **c** does not correspond to a character with length one in the initial shift state. Otherwise, it returns the single-byte representation of that character.

wctomb Subroutine

Purpose

Converts a wide character into a multibyte character.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <stdlib.h>
```

```
int wctomb ( Storage,  WideCharacter)
char *Storage;
wchar_t WideCharacter;
```

Description

The **wctomb** subroutine determines the number of bytes required to represent the wide character specified by the *WideCharacter* parameter as the corresponding multibyte character. It then converts the *WideCharacter* value to a multibyte character and stores the results in the area pointed to by the

Storage parameter. The **wctomb** subroutine can store a maximum of **MB_CUR_MAX** bytes in the area pointed to by the *Storage* parameter. Thus, the length of the area pointed to by the *Storage* parameter should be at least **MB_CUR_MAX** bytes. The **MB_CUR_MAX** macro is defined in the **stdlib.h** file.

Parameters

Item	Description
Storage	Points to an area where the result of the conversion is stored.
WideCharacter	Specifies a wide-character value.

Return Values

The **wctomb** subroutine returns a 0 if the *Storage* parameter is a null pointer. If the *WideCharacter* parameter does not correspond to a valid multibyte character, a -1 is returned. Otherwise, the number of bytes that comprise the multibyte character is returned.

wctrans, or wctrans_l Subroutine

Purpose

Define character mapping.

Library

Standard library (**libc.a**)

Syntax

```
#include <wctype.h>
```

```
wctrans_t wctrans (const char * charclass);
wctrans_t wctrans_l (const char * charclass, locale_t Locale);
```

Description

The **wctrans** and **wctrans_l** functions are defined for valid character mapping names identified in the current locale. The **charclass** is a string identifying a generic character mapping name for which codeset-specific information is required. The following character mapping names are defined in all locales "tolower" and "toupper".

The function returns a value of type **wctrans_t**, which can be used as the second argument to subsequent calls of **towctrans** and **towctrans_l**. The **wctrans** and **wctrans_l** functions determines values of **wctrans_t** according to the rules of the coded character set defined by character mapping information in the program's locale (category LC_CTYPE) or in the locale represented by *Locale*. The values returned by **wctrans** are valid until a call to **setlocale** that modifies the category LC_CTYPE.

The values returned by *wctrans_l()* function is valid only in calls to *wctrans_l()* function with a locale represented by *Locale* with the same LC_CTYPE category value.

Return Values

The **wctrans** and **wctrans_l** functions return 0 if the given character mapping name is not valid for the current locale (category LC_CTYPE), otherwise it returns a non-zero object of type **wctrans_t** that can be used in calls to **towctrans** and **towctrans_l**.

Error Codes

The **wctrans**, and **wctrans_l** function may fail if:

Item	Description
EINVAL	The character mapping name pointed to by charclass is not valid in the current locale.

wctype, wctype_l, or get_wctype Subroutine

Purpose

Obtains a handle for valid property names in the current locale for wide characters.

Library

Standard C library (**libc.a**).

Syntax

```
#include <wchar.h>
```

wctype_t wctype (*Property***)**
const char ****Property*;**

wctype_t get_wctype (*Property***)**
char ****Property*;**

wctype_t wctype_l (*Property***,** *Locale***)**
const char ****Property*;**
locale_t *Locale***;**

Description

The **wctype** and **wctype _l** subroutines obtain a handle for valid property names for wide characters as defined in the current locale or in the locale represented by *Locale* respectively. The handle is of data type **wctype_t** and can be used as the *WC_PROP* parameter in the **iswctype** and **iswctype_l** subroutine. Values returned by the **wctype** subroutine are valid until the **setlocale** subroutine modifies the **LC_CTYPE** category.

The values returned by the **wctype_l** subroutine is valid only in calls to the **iswctype_l** subroutine with a locale represented by *Locale* with the same **LC_CTYPE** category value.

The **get_wctype** subroutine is identical to the **wctype** subroutine.

The **wctype** subroutine adheres to X/Open Portability Guide Issue 5.

Parameters

Item	Description
Property	Points to a string that identifies a generic character class for which code set-specific information is required. The basic character classes are: **alnum** Alphanumeric character. **alpha** Alphabetic character. **blank** Space and tab characters. **cntrl** Control character. No characters in **alpha** or **print** are included. **digit** Numeric digit character. **graph** Graphic character for printing. Does not include the space character or **cntrl** characters, but does include all characters in **digit** and **punct**. **lower** Lowercase character. No characters in **cntrl**, **digit**, **punct**, or **space** are included. **print** Print character. Includes characters in **graph**, but does not include characters in **cntrl**. **punct** Punctuation character. No characters in **alpha**, **digit**, or **cntrl**, or the space character are included. **space** Space characters. **upper** Uppercase character. **xdigit** Hexadecimal character.
Locale	Specifies the locale in which character has to be converted.

Return Values

Item	Description
A value of type **wctype_t** (a handle for valid property names in the current locale)	Successful
-1	Unsuccessful (The *Property* parameter specifies a character class that is not valid for the current locale.)

wcwidth Subroutine

Purpose

Determines the display width of wide characters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>
```

int wcwidth (*WC*)

```
wchar_t WC;
```

Description

The **wcwidth** subroutine determines the number of display columns to be occupied by the wide character specified by the *WC* parameter. The **LC_CTYPE** subroutine affects the behavior of the **wcwidth** subroutine.

Parameters

Item	Description
WC	Specifies a wide character.

Return Values

The **wcwidth** subroutine returns the number of display columns to be occupied by the *WC* parameter. If the *WC* parameter is a wide character null, a value of 0 is returned. If the *WC* parameter points to an unusable wide character code, -1 is returned.

Examples

To find the display column width of a wide character, use the following:

```
#include <string.h>
#include <locale.h>
#include <stdlib.h>


main()
{
   wchar_t wc;
   int   retval;


   (void)setlocale(LC_ALL, "");
   /* Let wc be the wide character whose
   ** display width is to be found.
   */
   retval= wcwidth( wc );
   if(retval == -1){
            /*
            ** Error handling. Invalid wide character in wc.
            */
   }
}
```

wlm_assign Subroutine

Purpose

Manually assigns processes to a class or cancels prior manual assignments for processes.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_assign ( args)

struct wlm_assign *args;
```

Description

The **wlm_assign** subroutine:

- Assigns a set of processes specified by their process IDs (PIDS) or process group IDs (PGID) to a specified superclass or subclass, thus overriding the automatic class assignment or a prior manual assignment.
- Cancels a previous manual assignment for the specified processes, allowing the processes to be subjected to the automatic assignment rules again.

The target processes are identified by their process ID (pid) or by their process group ID (pgid). The **wlm_assign** subroutine allows specifying processes using a list of pids, a list of pgids, or both.

The name of a valid superclass or subclass must be specified to manually assign the target processes to a class. If the target class is a superclass, each process is assigned to one of the subclasses of the specified superclass according to the assignment rules for the subclasses of this superclass.

A manual assignment remains in effect (and a process remains in its manually assigned class) until:

- The process terminates.
- The Workload Manager (WLM) is stopped. When WLM is restarted, the manual assignments in effect when WLM was stopped are lost.
- The class the process has been assigned to is deleted.
- The manual assignment for the process is canceled.
- A new manual assignment overrides a prior one.

The name of a valid superclass or subclass must be specified to manually assign the target processes to a class. The assignment can be done or canceled at the superclass level, the subclass level, or both. The interactions between automatic assignment, inheritance and manual assignment are detailed in the Manual class assignment in Workload Manager in *Operating system and device management*.

Flags in the **wa_versflags** field described below are used to specify if the requested operation is an assignment or cancellation and at which level.

To assign a process to a class or cancel a prior manual assignment, the caller must have authority both on the process and on the target class. These constraints translate into the following:

- The root user can assign any process to any class.
- A user with administration privileges on the subclasses of a given superclass (that is, the user or group name matches the user or group names specified in the attributes **adminuser** and **admingroup** of the

superclass) can manually reassign any process from one of the subclasses of this superclass to another subclass of the superclass.

- A user can manually assign the user's own processes (same real or effective user ID) to a superclass or a subclass, for which the user has manual assignment privileges (that is, the user or group name matches the user or group names specified in the attributes **authuser** and **authgroup** of the superclass or the subclass).

This defines three levels of privilege among the persons who can manually assign processes to classes, root being the highest. For a user to modify or terminate a manual assignment, the user must be at the same level of privilege as the person who issued the last manual assignment, or higher.

Note: The **wlm_assign** subroutine works with the in-core WLM data structures. Even if the WLM current configuration is a set, it applies to the currently loaded regular configuration. If an assignment is made to a class that does not exist in all configurations of the set, it will be lost when the first configuration that does not contain this class is activated (when the class is deleted).

Parameter

Item	Description
args	Specifies the address of the **struct wlm_assign** data structure containing the parameters for the desired class assignment.

The following fields of the **wlm_args** structure and the embedded substructures can be provided:

Item	Description
wa_versflags	Needs to be initialized with **WLM_VERSION.** The flags values available, defined in the **sys/wlm.h** header file, are: • **WLM_ASSIGN_SUPER** • **WLM_ASSIGN_SUB** • **WLM_ASSIGN_BOTH** • **WLM_UNASSIGN_SUPER** • **WLM_UNASSIGN_SUB** • **WLM_UNASSIGN_BOTH**
wa_pids	Specifies the address of the array containing the process IDs of processes to be manually assigned. When this list is empty, a NULL pointer can be passed together with a count of zero (0).
wa_pid_count	Specifies the number of PIDS in the above array. Could be zero (0) if using only pgids to identify the processes.
wa_pgids	Specifies the address of the array containing the process group identifiers (pids) of processes to be manually assigned. When this list is empty, a NULL pointer can be passed together with a count of zero (0).
wa_pgid_count	Specifies the number of PGIDs in the above array. Could be zero (0) if using only pids to identify the processes. If both pids and pgids counts are zero (0), no process is assigned, but the operation is considered successful.

Item	Description
wa_classname	Specifies the full name of the superclass (**super_name**) or the subclass (**super_name.sub_name**) of the class you want to manually assign processes to. The class name field is ignored when canceling an existing manual assignment.

Return Values

Upon successful completion, the **wlm_assign** subroutine returns a value of 0. If the **wlm_assign** subroutine is unsuccessful, a non-0 value is returned. The routine is considered successful if some of the target processes are not found, (to account for process terminations) or are not assigned/deassigned due to a lack of privileges, for instance. If none of the processes in the lists can be assigned/deassigned, this is considered an error.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_assign_tag Subroutine

Purpose

Assigns a WLM tag to a set of processes or removes prior manual tag assignments for processes.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>
int wlm_assign_tag (args)
struct wlm_assign_tag * args;
```

Description

The **wlm_assign_tag** subroutine:

- Sets the Workload Manager (WLM) tag for a set of processes that are specified by their process identifiers (PIDs) or process group identifiers (PGID).
- Removes the WLM tag for a set of processes that are specified by their process identifiers (PIDs) or process group identifiers (PGIDs).

The target processes are identified by their PID or by their PGID. With the **wlm_assign_tag** subroutine, you specify the processes using a list of PIDs, a list of PGIDs, or both.

The WLM tag assignment remains in effect until the following events occur:

- The tag is removed using the **-r** flag.
- The tagged process ends.
- The tag is overwritten with a new tag.

When a WLM tag is assigned to a process and if the process is in a class with inheritance off, then the process is automatically reclassified according to the current assignment rules and the new tag is taken into account when doing this reclassification. The WLM tag is only effective if the current class

of the process does not have the class inheritance attribute specified. To override the class inheritance attribute in favor of reclassification based on tag rules, the **/usr/samples/kernel/wlmtune** command that is available in the **bos.adt.samples** PTF can be used to modify the behavior of WLM in such an instance. The related tunable are as follows:

tag_override_super
Indicates to WLM that superclass inheritance is bypassed in favor of a rule-based classification if there is a rule matching the process tag. The default value is 0.

tag_override_sub
Indicates to WLM that subclass inheritance is bypassed in favor of rule-based classification if there is a rule matching the process tag. The default value is 0.

The name of a valid superclass or subclass must be specified to manually assign the target processes to a class. The assignment can be done or canceled at the superclass level, the subclass level, or both. When a manual assignment is canceled for a process or the process calls the **exec()** system call, the process is then subject to automatic classification if inheritance is enabled for the class that the process is in, it will remain in that class; otherwise the process will be reclassified according to the assignment rules.

Parameter

Item	Description
args	Specifies the address of the **struct wlm_assign_tag** data structure that contains the parameters for the desired tag assignment.

The following fields of the **wlm_args** structure and the embedded substructures can be provided:

Item	Description
wt_versflags	Specifies the address of an integer that is interpreted in a manner similar to the **versflags** field of the **wlmargs** structure passed to other WLM APIs. The integer pointed to by flags must be initialized with the **WLM_VERSION** flag. In addition, one or more of the following values can be OR to the **WLM_VERSION** flag: **SWLMTAGINHERITFORK** Specifies that the children of this process inherit the parent tag on the **fork** subroutine. **SWLMTAGINHERITEXEC** Specifies that the process retains its tag after a call to the **exec** subroutine. Both flags can be set to specify that the children of a tagged process inherits the tag on the **fork** subroutine and then retains it on the **exec** subroutine.
wt_pids	Specifies the address of the array that contains the PIDs of the processes to be tagged. When this list is empty, a NULL pointer can be passed together with a count of 0.
wt_pid_count	Specifies the number of PIDs in the previous array. The number of PIDs will be 0 if only PGIDs are used to identify the processes.
wt_pgids	Specifies the address of the array containing the PGID of the processes to be tagged. When this list is empty, a NULL pointer can be passed together with a count of 0.
wt_pgid_count	Specifies the number of PGIDs in the above array. The number of PGID will be 0 if only PIDs are used to identify the processes. If both PID and PGID counts are 0, no processes are tagged, but the operation is considered successful.
wt_tagname	Specifies the full name of the WLM tag that you want to set for the processes. The maximum length of a tag name must not exceed 16 characters in length. An error is returned if this tag is too long. A NULL string will result in overwriting and effectively removing the process tag.

Return Values

Upon successful completion, the **wlm_assign_tag** subroutine returns a value of 0. If the **wlm_assign_tag** subroutine is unsuccessful, a nonzero value is returned. The routine is considered successful if some of the target processes are not found to account for process terminations. The **wlm_assign_tag** subroutine is considered successful when a tag name assignment or overwrite operation is performed on a process that contains a NULL tag attribute name.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_change_class Subroutine

Purpose

Changes some of the attributes of a class.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_change_class ( wlmargs)

struct wlm_args *wlmargs;
```

Description

The **wlm_change_class** subroutine changes attributes of an existing superclass or subclass. Except for its name, any of the attributes of the class can be modified by a call to **wlm_change_class**.

- If the name of a valid configuration is passed in the **confdir** field, the subroutine updates the Workload Manager (WLM) properties files for the target configuration.
- If a null string ('\0') is passed in the **confdir** field, the changes are applied only to the in-core WLM data. No WLM properties file is updated.

The structure of type **struct class_definition**, which is part of **struct wlm_args**, has normally been initialized with a call to **wlm_init_class_definition**. Once this has been done, initialize the required fields of this structure (such as the name of the class to be modified) and the fields corresponding to the class attributes you want to modify. For a description of the possible values for the various class attributes and their default values, refer to the description of **wlm.h** in the *Files Reference*.

The caller must have root authority to change the attributes of a superclass and must have administrator authority on a superclass to change the attributes of a subclass of the superclass.

Note: Do not specify a set in the *confdir* field of the **wlm_args** structure. The **wlm_change_class** subroutine cannot apply to a set of time-based configurations.

Parameters

Item	Description
wlmargs	Specifies the address of the **struct wlm_args** data structure containing the **class_definition** structure for the class to be modified.

The following fields of the **wlm_args** structure and the embedded substructures need to be provided:

Item	Description
versflags	Needs to be initialized with **WLM_VERSION.**
confdir	Specifies the name of the WLM configuration the target class belongs to. It must be either the name of a valid subdirectory of **/etc/wlm** or an empty string (starting with '\0'). If the name is a valid subdirectory, the relevant class description file in the given configuration are modified. If the name is a null string, no description files are updated. The modified class attributes are passed to the kernel similarly to a call to **wlm_load**.
name	Specifies the name of the superclass or of the subclass to be modified. If this is a subclass name, it must be of the form **super_name.sub_name**. There is no default for this field.

All the other fields can be left at their initial value as set by **wlm_init_class_definition** if the user does not wish to change the current values.

Return Values

Upon successful completion, the **wlm_change_class** subroutine returns a value of 0. If the **wlm_change_class** subroutine is unsuccessful, a nonzero value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_check subroutine

Purpose

Check a WLM configuration.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_check ( config)

char *config;
```

Description

The **wlm_check** subroutine checks the class definitions and the coherency of the assignment rules file(s) (syntax, existence of the classes, validity of user and group names, application path names, etc.) for the configuration whose name is passed as an argument.

If *config* is a null pointer or points to an empty string, **wlm_check** performs the checks on the configuration files, in the configuration pointed to by **/etc/wlm/current**.

The **wlm_check** subroutine can apply to a configuration set. If *config* is a configuration set name (or if *config* is not provided and *current* is a configuration set), the checks mentioned above are performed on all configurations of the set, after checking the set itself.

Parameter

Item	Description
config	A pointer to a character string. This pointer should be: • The address of a character string representing the name of a valid configuration (a subdirectory of **/etc/wlm**) • A null pointer • A pointer to a null string ("") If *config* is a null pointer or a pointer to a null string, the configuration files in the directory pointed to by **/etc/wlm/current** (active configuration) is checked for errors. Otherwise, the configuration files in directory /**etc/wlm/<config_name>** is checked.

Return Values

Upon successful completion, a value of 0 is returned. If the **wlm_check** subroutine is unsuccessful a non 0 value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the header file **sys/wlm.h**.

wlm_classify Subroutine

Purpose

Determines which classes a process is assigned to.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_classify ( config, attributes, class, len)

char *config;

char *attributes;

char *class;

int *len;
```

Description

The **wlm_classify** subroutine must be passed the name of a valid configuration and a set of process *attributes* in a format identical to the format of the **rules** file (assignment rules). The names of the classes

are copied into the area pointed to by *class*. The integer pointed to by *len* contains the size of the *class* names area on input and the number of matches on output. If the area pointed to by *class* is not big enough to contain the names of all the potential matches, an error is returned.

The normal use of the **wlm_classify** routine is to explicitly provide all the process classification attributes: **user name**, **group name**, **application pathname**, **type**, and **tag** when applicable. This gives a match to a single class. To implement "what if" scenarios, the interface allows you to leave some of the attributes unspecified by using a hyphen ('-') instead. This may lead to multiple classes the process could be assigned to, depending on the values of the unspecified attributes. If all the attributes are left unspecified, an error is returned.

The *attributes* string is provided in a format identical to the format of the attributes in the rules file: a list of attribute values separated by spaces. The order of the attributes in the assignment rules is:

1. reserved: must be a hyphen ('-')
2. user name
3. group name
4. application pathname
5. type of application
6. tag

Each field can have at most one value. Exclusion (!), attribute value groupings ($), comma separated lists and wild cards are not allowed. For the type field, the AND operator "+" is allowed, since a process can have several of the possible values for the type attribute at the same time. For instance a process can be a 32 bit process and call plock, or be a 64 bit fixed priority process.

Here are examples of valid *attributes* strings:

```
"- bob staff /usr/bin/emacs - -"
"- - - /usr/sbin/dbserv - _DB1"
"- - devlt - 32bit+fixed"
"- sally"
```

The class name(s) returned by the function in the *class* buffer is fully-qualified, null-terminated class names of the form **supername.subname**.

This function does not require any special privileges and can be called by all users.

Parameters

Item	Description
config	Specifies a pointer to a string containing the name of a valid Workload Manager (WLM) configuration (the name of a subdirectory of **/etc/wlm**). If a null string ('\0') is given, the **wlm_classify** subroutine uses *current* as the default configuration. If the configuration is a set of time-based configurations, either because *config* or *current* is a configuration set, the subroutine will apply to the currently applicable configurations of the set.
attributes	Specifies the address of a string, with the format described above, containing a list of values for the process attributes used for automatic classification of processes.
class	Specifies a pointer to a buffer where the name of the class the process could be assigned to is returned as consecutive null-terminated character strings.

Item	Description
len	Specifies a pointer to an integer containing the length in bytes of the buffer pointed to by *class* when calling **wlm_classify** and the actual number of class names copied into the *class* buffer upon successful return.

Return Values

Upon successful completion, the **wlm_classify** subroutine returns a value of 0. In case of error, a non-0 value is returned.

When a non-0 value is returned, the content of the **class** buffer and the value of the integer pointed to by **len** are unspecified.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_class2key Subroutine

Purpose

Class name to key translation.

Library

Workload Manager Library (**libwlm.a**)

Syntax

#include <sys/wlm.h

int wlm_class2key (struct wlm_args *args, wlm_key_t *key)

Description

The **wlm_class2key** subroutine generates a 64-bit numeric key from a WLM class name. The **wlm_class2key** subroutine is provided for applications gathering high volumes of per-class usage statistics or accounting data and allows those applications to save storage space by compressing the class name (up to 34 characters long) into a 64-bit integer. The **wlm_key2class** subroutine can then get the key-to-class name conversion for data reporting purposes

Parameters

Item	Description
wlm_args	Only 2 fields need to be initialized in the **wlm_args** structure pointed to by **args**: • *cl_def.data.descr.name* specifies the null terminated full name of the class (<super_name.<subname for a subclass). • *versflags* initialized with WLM_VERSION and optionally WLM_MUTE.

Return Values

If the **wlm_class2key** subroutine is successful, a value of 0 is returned. If the **wlm_class2key** subroutine is unsuccessful, an error code is returned.

Error Codes

If the **wlm_class2key** subroutine is unsuccessful, one of the following error codes is returned:

Item	Description
WLM_NOT_INITED	Missing call to **wlm_init**.
WLM_EFAULT	Invalid key or args pointer.
WLM_BADCNAME	The class name contains invalid characters.

wlm_create_class Subroutine

Purpose

Creates a new Workload Manager (WLM) class.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_create_class ( wlmargs)

struct wlm_args *wlmargs;
```

Description

The **wlm_create_class** subroutine creates a new class for a given WLM configuration using the values passed in the data structure of type **struct wlm_args** pointed to by *wlmargs.*

- If the name of a configuration is passed in the **confdir** field, the subroutine updates the WLM properties files for the target configuration. When creating the first subclass of a superclass, the subroutine creates a subdirectory of **/etc/wlm/<confdir>** with the name of the superclass and create the WLM properties files in this new directory. The newly created properties files have entries for the Default and Shared subclass automatically created in addition to entries for the new subclass.
- If a null string ('\0') is passed in the **confdir** field, the new superclass or subclass is created only in the in-core WLM data. No WLM properties file are updated. In that case, the new class definition is lost if WLM is stopped and restarted, or if the system reboots.

The structure of type **struct class_definition**, which is part of **struct wlm_args**, has normally been initialized with a call to **wlm_init_class_definition**. Once this has been done, initialize the fields of this structure which have no default value (such as the name of the new class) or for which the desired value is different from the default value. For a description of the possible values for all the class attributes and their default values, refer to the description of **wlm.h** in the *Files Reference*.

The caller must have root authority to create a superclass and must have administrator authority on a superclass to create a subclass of the superclass.

Note: Do not specify a set in the *confdir* field of the **wlm_args** structure. The **wlm_create_class** subroutine cannot apply to a set of time-based configurations.

Parameter

Item	Description
wlmargs	Specifies the address of the **struct wlm_args** data structure containing the **class_definition** structure for the new class to be created.

The following fields of the **wlm_args** structure and the embedded substructures need to be provided:

Item	Description
versflags	Needs to be initialized with **WLM_VERSION.**
confdir	Specifies the name of the WLM configuration the new class is to be added to. It must be either the name of a valid subdirectory of **/etc/wlm** or an empty string (starting with '\0'). If the name is a valid subdirectory, the new class data is added to the given WLM configuration's class description files. If the name is a null string, no description files are updated. The new class is created and the data is passed to the kernel immediately.
name	Specifies the name of the superclass or of the subclass to be created. If this is a subclass name, it must be of the form **super_name.sub_name**. There is no default for this field.

All the other fields can be left at their default value if the user does not wish to use specific values.

Return Values

Upon successful completion, the **wlm_create_class** subroutine returns a value of 0. If the **wlm_create_class** subroutine is unsuccessful, a nonzero value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_delete_class Subroutine

Purpose

Deletes a class.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_delete_class ( wlmargs)

struct wlm_args *wlmargs;
```

Description

The **wlm_delete_class** subroutine deletes an existing superclass or subclass. A superclass cannot be deleted if it still has subclasses other than Default and Shared defined.

- If the name of a valid configuration is passed in the **confdir** field, the subroutine updates the Workload Manager (WLM) properties files for the target configuration, removing all references to the class to be deleted.
- If a null string ('\0') is passed in the **confdir** field, the class is deleted only from the in-core WLM data structures. No WLM properties file is updated. This is normally used to delete a class which was also only created in the in-core WLM data structures. Otherwise, the class deletion is temporary and the class will be created again when WLM is updated or restarted with a configuration where the class exists in the classes file.

The caller must have root authority to delete a superclass and must have administrator authority on a superclass to delete a subclass of the superclass.

Note: Do not specify a set in the *confdir* field of the **wlm_args** structure. The **wlm_delete_class** subroutine cannot apply to a set of time-based configurations.

Parameter

Item	Description
wlmargs	Specifies the address of the **struct wlm_args** data structure containing the information about the class to be deleted.

The following fields of the **wlm_args** structure and the embedded substructures need to be provided:

Item	Description
versflags	Needs to be initialized with **WLM_VERSION**.
confdir	Specifies the name of the WLM configuration the target class belongs to. It must be either the name of a valid subdirectory of **/etc/wlm** or an empty string (starting with '\0'). If the name is a valid subdirectory, the relevant class description files in the specified configuration are modified. If the name is a null string, no description files are updated. The class is removed from the kernel WLM data structures.
name	Specifies the name of the superclass or of the subclass to be deleted. If this is a subclass name, it must be of the form **super_name.sub_name**. There is no default for this field.

All the other fields can be left uninitialized for this call.

Return Values

Upon successful completion, the **wlm_delete_class** subroutine returns a value of 0. If the **wlm_delete_class** subroutine is unsuccessful, a non-0 value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_endkey Subroutine

Purpose

Frees the classes to keys translation table.

Library

Workload Manager Library (**libwlm.a**)

Syntax

#include sys/wlm.h

int wlm_endkey(struct wlm_args *args, void *ctx)

Description

The **wlm_endkey** subroutine frees the classes to the keys translation table. The memory area pointed to by *ctx* is freed.

Parameters

Item	Description
- *ctx*	Points to the memory area to be freed.
wlm_args	A pointer to a **wlm_args** structure: **versflag** field is the only field in the structure that needs to be initialized with WLM_VERSION and optionally WLM_MUTE.

Return Values

When the **wlm_endkey** operation is successful, it returns a value of 0, and if it is unsuccessful, it returns an error code.

Error Codes

If the **wlm_endkey** subroutine is unsuccessful, one of the following error codes is returned:

Item	Description
WLM_BADVERS	Bad version number.
WLM_NOT_INITED	Missing call to **wlm_init**.
WLM_EFAULT	Invalid *ctx* or *args* argument.

wlm_get_bio_stats subroutine

Purpose

Read the WLM disk I/O statistics per class or per device.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/types.h>

#include <sys/wlm.h>

int wlm_get_bio_stats ( dev, array, count, class, flags)

dev_t dev;

void *array;

int *count;

char *class;

int flags;
```

Description

The **wlm_get_bio_stats** subroutine is used to get the WLM disk IO statistics. There are two types of statistics available:

- The statistics about disk IO utilization per class and per devices, returned by **wlm_get_bio_stats** in **wlm_bio_class_info_t** structures,
- The statistics about the disk IO utilization per device, all classes combined, returned by **wlm_get_bio_stats** in **wlm_bio_dev_info_t** structures.

The type of statistics returned by the function is predicated on the value of the *flags* argument. The *flags* argument, together with the *dev* and *class* arguments, are used to restrict the scope of the function to a class or a set of classes and/or a device or a set of devices. If the value passed to the routine in the *count* argument is equal to zero (0), **wlm_get_bio_stats** does not copy any device statistics (and, in this case, the *array* argument can be a NULL pointer but sets this count to the number of elements in scope for the specific set of parameters. This is a way of finding out how big an array is needed to get all the information for a given set of classes and devices.

wlm_get_bio_stats does not require any special privileges and is accessible to all users. **wlm_get_bio_stats** fails if WLM is off.

Parameters

Item	Description
flags	Need to be initialized with **WLM_VERSION.** Optionally, the following flag values can be or'ed to **WLM_VERSION:** **WLM_SUPER_ONLY** Limits the scope to superclasses only **WLM_SUB_ONLY** Limits the scope to subclasses only **WLM_BIO_CLASS_INFO** Per class statistics requested **WLM_BIO_DEV_INFO** Per device statistics requested **WLM_BIO_ALL_DEV** Requests statistics for all devices. When this flag is set, the value passed in the *dev* argument is ignored. **WLM_BIO_ALL_MINOR** Requests statistics for all devices associated with a given major number. When this flag is set, only the major number part of the value passed in the *dev* argument is used. **WLM_VERBOSE_MODE** Shows the system defined subclasses (*Default* and *Shared)*even if they have not been modified by a WLM administrator. One of the flags **WLM_BIO_CLASS_INFO** or **WLM_BIO_DEV_INFO** (and only one) must be specified. **WLM_SUPER_ONLY** and **WLM_SUB_ONLY**are mutually exclusive.
dev	Device identification (major, minor) of a disk device. • If *dev* is equal to 0, the statistics for all devices are returned (even if **WLM_BIO_ALL_DEV** is not specified in the *flags* argument). • If *dev* is not equal to 0 and **WLM_BIO_ALL_MINOR** is specified in the *flags* argument, the statistics for all disk devices with the same major number specified in *dev* are returned. • If *dev* is not equal to 0 and **WLM_BIO_ALL_MINOR** is not specified in the *flags* argument, only the statistics for the disk device with the major and minor numbers specified in *dev* are returned.

Item	Description
array	Pointer to an array of **wlm_bio_class_info_t** structures (when **WLM_BIO_CLASS_INFO** is specified in the *flags* argument) or an array of **wlm_bio_dev_info_t** structures (when **WLM_BIO_DEV_INFO** is specified in the *flags* argument). A NULL pointer can be passed together with a *count* of 0 to determine how many elements are in scope for the set of arguments passed.
count	The address of an integer containing the maximum number of elements to be copied into the array above. If the call to **wlm_get_bio_stats** is successful, this integer will contain the number of elements actually copied. If the initial value is equal to zero (0), **wlm_get_bio_stats** sets this value to the number elements selected by the specified combination of flags and class.
class	A pointer to a character string containing the name of a superclass or subclass. If class is a pointer to an empty string (""), the information for all classes are returned. The *class* parameter is taken into account only when the flag **WLM_BIO_CLASS_INFO** is set.

Return Values

Upon successful completion, a value of 0 is returned and the value pointed to by *count* is set to the number of elements copied into the array of structures pointed to by *array*. If the **wlm_get_bio_stats** subroutine is unsuccessful a non 0 value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the header file **sys/wlm.h**.

wlm_get_info Subroutine

Purpose

Read the characteristics of superclasses or subclasses.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_get_info ( wlmargs,  info,  count)

struct wlm_args *wlmargs;

struct wlm_info *info
```

int *count

Description

The **wlm_get_info** subroutine is used to get the characteristics of the classes defined in the active Workload Manager (WLM) configuration, together with their current resource usage statistics. For a detailed description of the fields of the structure **wlm_info**, refer to the description of the **wlm.h** header file in the *Files Reference* documentation.

By default, the scope of the **wlm_get_info** subroutine is all the superclasses and all the subclasses. This scope can be limited to a subset of the classes using flags in the **versflags** field of **wlm_args** or a superclass or subclass name in the **name** field of the substructure **class_definition** of **wlm_args**.

The information related to the superclasses and subclasses within the scope of **wlm_get_info** are copied to the array of **wlm_info** structures pointed to by *info*. The total number of classes for which information is copied to the array at *info* is limited to the value of the integer pointed to by *count*. If the routine is successful, the value of the integer pointed to by count is set to the actual number of classes copied. If the value passed to the routine for the count is equal to zero (0), **wlm_get_info** does not copy any class statistics but sets this count to the number of classes in scope for the specific set of parameters. This is a way of finding out how big an array is needed to get all the information for a given set of classes (superclasses or subclasses).

This is a way of finding out how big an array is needed to get all the information for a given set of classes (superclasses or subclasses).

The **wlm_get_info** subroutine does not require any special privileges and is accessible to all users. **wlm_get_info** fails if WLM is off.

Parameters

wlmargs
The address of a **struct wlm_args** data structure.

The following fields of the **wlm_args** structure and the embedded substructures need to be provided:

versflags
Needs to be initialized with **WLM_VERSION**. Optionally, the following flag value can be or'ed to **WLM_VERSION:**

WLM_SUPER_ONLY
Limits the scope to superclasses only

WLM_SUB_ONLY
Limits the scope to subclasses only

WLM_VERBOSE_MODE
Shows the system-defined subclasses (Default and Shared) even if they have not been modified by a WLM administrator.

WLM_SUPER_ONLY and **WLM_SUB_ONLY** are mutually exclusive.

name
Contains either a null string or the name of a valid superclass or subclass (in the form **Super.Sub**). This field can be used in conjunction with the flags to further narrow the scope of **wlm_get_info**:

- If the name of a subclass is provided, **wlm_get_info** returns the statistics only for the specified subclass.
- If the name of a superclass is provided or if none of the **WLM_SUPER_ONLY** and **WLM_SUB_ONLY** flag is provided, **wlm_get_info** returns the statistics for the specified superclass and all its subclasses.

- If the name of a superclass is provided together with **WLM_SUPER_ONLY**, **wlm_get_info** returns only the statistics for the specified superclass.
- If the name of a superclass is provided together with **WLM_SUB_ONLY**, **wlm_get_info** returns the statistics for all the subclasses of the specified superclass.

All the other fields of the **wlm_args** structure can be left uninitialized.

info
The address of an array of structures of type **struct wlm_info**. Upon successful return from **wlm_get_info**, this array contains the WLM statistics for the classes selected.

count
The address of an integer containing the maximum number of element (of type **wlm_info**) for **wlm_get_info** to copy into the array above. If the call to **wlm_get_info** is successful, this integer contains the number of elements actually copied. If the initial value is equal to zero (0), **wlm_get_info** sets this value to the number of classes selected by the specified combination of **versflags** and **name** above.

Return Values

Upon successful completion, the **wlm_get_info** subroutine returns a value of 0. If the **wlm_get_info** subroutine is unsuccessful a non-0 value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_get_procinfo Subroutine

Purpose

Retreives per-process Workload Manager information.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_get_procinfo (pid, wlmpinfop)
pid_t pid;
struct wlm_procinfo *wlmpinfop;
```

Description

The **wlm_get_procinfo** subroutine returns Workload Manager information for the process associated with the *pid* parameter, into the buffer pointed to by the *wlmpinfop* parameter. If process total accounting is disabled, the related fields (*totalconnecttime*, *termtime*, *totalcputime*, and *totaldiskio*) are set to -1. When WLM is on, the class name of the process is set in the *classname* field of the **wlm_procinfo** structure. When WLM is off, this field is set to *Unclassified*.

Parameters

Item	Description
pid	Indicates from which process to retrieve the Workload Manager information.

Item	Description
wlmpinfop	Points to the buffer where the Workload Manager information is stored.
wlminfop	The address of a **struct wlm_procinfo** data structure. The following fields of the **wlm_procinfo** structure need to be provided: *version* Needs to be initialized with **WLM_VERSION**.

Return Values

Upon successful completion, the **wlm_get_procinfo** subroutine returns a zero. If the **wlm_get_procinfo** subroutine is unsuccessful, a nonzero value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_init_class_definition Subroutine

Purpose

Initializes a variable of type **struct class_definition**, defined in **<sys/wlm.h>** for use as an argument to Workload Manager (WLM) API function calls.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_init_class_definition ( wlmargs)

struct wlm_args *wlmargs;
```

Description

The **wlm_init_class_definition** subroutine initializes or reinitializes the data structure of type **struct class_definition**, which is part of the argument of type **struct wlm_args** pointed to by *wlmargs* (field **class**), so that this data structure can be used as an argument for the class management subroutines of the WLM API library. The purpose of this call is to allow applications to initialize only the fields that are relevant for the operation they execute. For example, to change a CPU limit or share for an existing class after a call to **wlm_init_class_definition**, the application has to initialize the fields corresponding to the values it wishes to modify.

This routine initializes all values to specific invalid values so that the WLM library routines can find out which fields have been explicitly initialized by the user. This way, they can set or modify only the corresponding attributes. When creating a class, for instance, it is different to leave a **class** attribute at its invalid value set by **wlm_initialize** than setting its value to the current default value for the attribute. In the former case, the attribute will not appear in the property file. In the latter, it will appear and will be set with the value passed.

This makes a difference if a WLM administrator decides to change the default value for an attribute using the special stanza default in a property file. For instance, the system default for the **inheritance** attribute

is no. If a WLM administrator wants the inheritance to be yes by default, using this special stanza, all the classes in the classes property file, for which the **inheritance** attribute has not been specified, will now use the default of yes. Those for which the **inheritance** attribute has been specified with its old default of no will not have inheritance.

Parameter

Item	Description
wlmargs	Specifies the address of the **struct wlm_args** data structure containing the **class_definition** structure to be initialized.

Only the **versflags** field of the **wlm_args** structure passed need to be initialized with **WLM_VERSION**.

Return Values

Upon successful completion, the **wlm_init_class_definition** subroutine returns a value of 0. If the **wlm_init_class_definition** subroutine is unsuccessful a non-0 value is returned.

Error Codes

There are two possible error code returned by **wlm_init_class_definition**:

Item	Description
BADVERSION	Specifies the value of the flags parameter is not a supported version number.
NOTINITED	Specifies the WLM API has not been initialized by a prior call to **wlm_init**.

wlm_initialize Subroutine

Purpose

Prepares Workload Manager (WLM) for use by an application.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_initialize ( flags)

int flags;
```

Description

The **wlm_initialize** subroutine initializes the WLM API for use with an application program. It is mandatory to call **wlm_initialize** prior to using the WLM API. Otherwise, all other WLM API function calls return an error.

Parameter

Item	Description
flags	Specifies that the format is the same as the **versflag** field of the **wlm_args** structure. The value for the argument must have the version number in the upper 4 bits (**WLM_VERSION**) possibly or'ed with a flag in the lower 28 bits.

Return Values

Upon successful completion, the **wlm_initialize** subroutine returns a value of 0. If the **wlm_initialize** subroutine is unsuccessful a non-0 value is returned.

Error Codes

There are two possible error codes returned by **wlm_initialize**:

Item	Description
BADVERSION	The value of the *flags* parameter is not a supported version number.
WLMINITED	There has already been a previous call to **wlm_initialize**.

wlm_initkey Subroutine

Purpose

Allocates and initializes the classes to keys translation table.

Library

Workload Manager Library (**libwlm.a**)

Syntax

#include <sys/wlm.h

int wlm_initkey (struct wlm_args *args, void **ctx)

Description

The **wlm_initkey** subroutine allocates a block of memory, builds the keys == class names translation table and returns its address into the **ctx** argument.

Parameters

Item	Description
args	Only 2 fields need to be initialized in the **wlm_args** structure pointed to by **args**: • *confdir* specifies the null-terminated name of the WLM configuration to be searched (the name can be "current" to specify the current configuration). If the configuration name passed is an empty string (starts with '\0'), then all the configurations in **/etc/wlm** are searched. • *versflags* initialized with WLM_VERSION and optionally WLM_MUTE.

Return Values

If the **wlm_initkey** subroutine is successful, a value of 0 is returned. If the **wlm_initkey** subroutine is unsuccessful, an error code is returned.

Error Codes

If the **wlm_initkey** subroutine is unsuccessful, one of the following error codes is returned:

Item	Description
WLM_BADVERS	Bad version number.
WLM_NOT_INITED	Missing call to **wlm_init**.
WLM_NOMEM	Not enough memory.
WLM_NOCLASS	Specified configuration does not exist.
WLM_EFAULT	Invalid **ctx** or **args** argument.

wlm_key2class Subroutine

Purpose

Retrieves a class name from a key.

Library

Workload Manager Library (**libwlm.a**)

Syntax

#include <sys/wlm.h

int wlm_key2class (struct wlm_args *args, wlm_key_t key, void *ctx)

Description

The **wlm_key2class** subroutine retrieves a class name from a 64-bit key calculated using the **wlm_class2key** subroutine. The key-to-class translation is made by going through the WLM configuration files for the configuration named in the **wlm_args** structure pointed to by args (or all the WLM configuration files, if no configuration name is given), and translating all the class names to a 64-bit key until the matching key is found.

This process is time consuming and WLM offers the subroutines **wlm_initkey** and **wlm_endkey** for applications needing to translate several 64-bit keys back to class names. These subroutines can be used in conjunction with the **wlm_key2class** subroutine to speed up searches.

The **wlm_initkey** subroutine allocates a block of memory, calculates the keys corresponding to the class names in the configuration(s) in scope, stores the names with the corresponding keys in the memory buffer, and returns its address. This address is passed to the **wlm_key2class** subroutine using the **ctx** argument, so that **wlm_key2class** only needs to search through the memory buffer.

After all keys have been translated into class names, the application must call **wlm_endkey** to free the memory buffer. Alternatively, for an application translating only one key, it is possible to call **wlm_key2class** directly using a null pointer in the **ctx** argument. This causes the **wlm_key2class** subroutine to internally call **wlm_initkey** and **wlm_endkey**.

The method of retrieving class names through the WLM configuration files implies that if a class has been deleted between the time the class name was converted into a key and the call to the **wlm_key2class**

subroutine, the name corresponding to the key will not be found and the **wlm_key2class** subroutine returns an error.

Parameters

Item	Description
- args	A pointer to a **wlm_args** structure: • **confdir** field needs to be initialized as described in **wlm_initkey** if **wlm_initkey** has not been previously invoked (ctx == NULL). Otherwise, the **confdir** field is ignored. • **versflags** field needs to be initialized with WLM_VERSION and optionally WLM_MUTE.
- ctx	The context handler returned by **wlm_initkey**, or a NULL pointer otherwise. .
- key	The search key.

Return Values

When the **wlm_key2class** operation is successful, the first class name matching the value of the key is returned in the name sub-field of the **wlm_args** structure pointed to by **args**.

Error Codes

If the **wlm_key2class** subroutine is unsuccessful, one of the following error codes is returned:

Item	Description
WLM_BADVERS	Bad version number.
WLM_NOT_INITED	Missing call to **wlm_init**.
WLM_NOMEM	Not enough memory.
WLM_NOCLASS	No class matching the key was found.
WLM_EFAULT	Invalid *ctx* or *args* argument.

wlm_load Subroutine

Purpose

Loads a Workload Manager (WLM) configuration into the kernel.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_load ( wlmargs)

struct wlm_args *wlmargs;
```

Description

The **wlm_load** subroutine loads into the kernel the property files for the WLM configuration passed in the *confdir* field of the **wlmargs** structure. The *confdir* field may also refer to a set of time-based configurations, in which case the appropriate configuration of the set will be loaded and the WLM daemon will later switch to the other configurations of the set on a time basis.

If the WLM is running and *confdir* is not current, this leads to switch to the specified configuration (or configuration set).

If the WLM is running and *confdir* is current, **wlm_load** will refresh the current WLM configuration into the kernel. If a superclass name is given in the *name* field of the class_definition substructure, only the subclasses of the given superclass are refreshed. In this context:

- The **wlm_load** subroutine is accessible to root users and to users with administration privileges on the subclasses of the superclass. In all other cases, the **wlm_load** subroutine is only accessible to root users.
- The **wlm_load** subroutine cannot be used to change the mode of operation of WLM (for example, to switch between active and passive modes).
- If *current* is a configuration set, *confdir* must be given in the form *current*/*config* where *config* is the regular configuration of the set the superclass belongs to. If *config* is the active configuration of the set, the changes will take effect immediately, otherwise they will take effect the next time *config* is made active.

If the caller of **wlm_load** has root privileges and does not specify a superclass, the flags passed in *versflags* can be used to start WLM in active or passive mode, switch between active and passive modes, or enable/disable the rset bindings or the process or class total limits. The **wlm_load** subroutine cannot be used to stop WLM. Use the **wlm_set** subroutine instead.

Parameter

Item	Description
wlmargs	Specifies the address of the **struct wlm_args** data structure containing information about the configuration (or configuration set or superclass) to be loaded and the mode of operation of WLM.

The following fields of the **wlm_args** structure and the embedded substructures can be provided:

Item	Description
versflags	Needs to be initialized with WLM_VERSION. May be ORed with WLM_MUTE for **wlm_load** to be silent. If no change must be done to the mode of operation of WLM, it must be ORed with WLM_TEST_ON (mandatory if superclass is specified). Otherwise, one of the mutually exclusive flags (WLM_ACTIVE, WLM_CPUONLY, or WLM_PASSIVE) must be given. One or more of the WLM_BIND_RSETS, WLM_PROCTOTAL, or WLM_CLASSTOTAL flags can be given optionally.
confdir	Specifies the name of the WLM configuration to be loaded into the kernel. It must be either the name of a valid configuration or configuration set in the **/etc/wlm** subdirectory, the *current* string to refer to the active configuration, or, if superclass is specified and current is a configuration set, it must indicate which configuration of current set the superclass belongs to in the form: *current*/*config* (this is different from specifying *config* only, which is considered a configuration switch request).
name	Specifies the name of a **superclass**. This is used to refresh only the subclasses of a given superclass.

Return Values

Upon successful completion, the **wlm_load** subroutine returns a value of 0. If the **wlm_load** subroutine is unsuccessful, a nonzero value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_read_classes Subroutine

Purpose

Reads the characteristics of superclasses or subclasses.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_read_classes (wlmargs, class_tbl, nclass)
struct wlm_args *wlmargs;
struct class_definition *class_tbl;
int *nclass;
```

Description

The **wlm_read_classes** subroutine is used to get the characteristics of the superclasses or the subclasses of a given subclass of a Workload Manager (WLM) configuration.

- If the name of a configuration is passed in the **confdir** field, the **wlm_read_classes** subroutine reads the property files of the classes of the specified configuration. If **confdir** is set to a null string ('\0'), **wlm_read classes** reads the classes' characteristics from the in-core WLM data structures when WLM is on (and returns an error when WLM is off).

  Note: These values may be different from the values in the property files of the configuration pointed to by **/etc/wlm/current**. For instance when a WLM administrator has modified the property files for the configuration pointed to by **/etc/wlm/current** but has not refreshed WLM yet. Another example is if applications dynamically created or modified classes through the API without saving the changes in the *current* configuration property files.

  If your application specifically needs to access the properties of the classes as described in the **/etc/wlm/current** configuration, you must specify *current* as the configuration name in **confdir**.

  If the name of a set of time-based configurations is passed in the *confdir* field, the **wlm_read_classes** subroutine reads the classes of the currently applicable configuration of the set.
- If the name of a valid superclass of the given configuration is passed in the **name** field of the **class_descr** substructure of *wlmargs*, **wlm_read_classes** reads the property files for the subclasses of this superclass. If a null string ('\0') is passed in the **name** field, **wlm_read_classes** reads the property files for the superclasses of the WLM configuration described above.
- When **wlm_read_classes** is successful, the characteristics of the superclasses or subclasses are copied into the array of **class_definition** structures pointed to by *class_tbl*. The integer value pointed to by *nclass* indicates the maximum number of class definitions to be copied. Upon successful return from the function, this value reflects the actual number of classes read.

If the number of elements copied by **wlm_read_classes** is strictly smaller than the number of elements passed as an argument, all the classes have been read. If it is equal, it may mean that some classes were not copied into the **class_tbl** array because its size is too small.

The maximum number of classes read by **wlm_read_classes** is 67 (64 user-defined superclasses plus System, Shared and Default) when reading superclasses and 63 (61 user-defined subclasses plus Shared and Default) when reading subclasses characteristics.

- Upon successful return from **wlm_read_classes**, the substructure **class** of type **struct class_definition** of the structure pointed to by *wlmargs* contains the default values of various class attributes for the returned set of classes.

This operation does not require any special privileges and is accessible to all users.

Parameter

Item	Description
wlmargs	Specifies the address of a **struct wlm_args** data structure. The following fields of the **wlm_args** structure and the embedded substructures need to be provided: **versflags** Needs to be initialized with **WLM_VERSION**. **confdir** Specifies the name of a WLM configuration. It must be either the name of a valid subdirectory of **/etc/wlm** or a null string (starting with '\0'). **name** Specifies the name of a superclass existing in the specified configuration or a null string. All the other fields can be left uninitialized.

Item	Description
class_tbl	Specifies the address of an array of structures of type **struct class_definition**. Upon successful return from **wlm_read_classes**, this array contains the characteristics of the classes read.
nclass	Specifies the address of an integer containing the maximum number of element (class definitions) for **wlm_read_classes** to copy into the array above. If the call to **wlm_read_classes** is successful, this integer contains the number of elements actually copied.

Return Values

Upon successful completion, the **wlm_read_classes** subroutine returns a value of 0. If the **wlm_read_classes** subroutine is unsuccessful, a nonzero value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_set Subroutine

Purpose

Sets or queries the Workload Manager (WLM) state.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_set ( flags)
int  *flags;
```

Description

The **wlm_set** subroutine is used to set, change, or query the mode of operations of WLM. The state of WLM can be:

Item	Description
OFF	Does not classify processes, monitor or regulate resource utilization.
ON in **passive** mode	Classifies the processes and monitors their resource usage but does no regulation.
ON in **active** mode	Specifies the normal operating mode where WLM classifies processes, monitors and regulates the resource usage.

Parameters

Item **Description**

flags Specifies the address of an integer interpreted in a manner similar to the **versflags** field of the **wlmargs** structure passed to the other API routines. The integer pointed to by *flags* should be initialized with **WLM_VERSION**. In addition, one or more of the following values can be or'ed to **WLM_VERSION**:

WLM_TEST_ON
Queries the state of WLM without altering it.

WLM_OFF
Turns WLM off.

WLM_ACTIVE
Turns WLM on in **active** mode or transitions from any mode to **active** mode.

WLM_CPU_ONLY
Turns WLM on in **active** mode for CPU resource only, or transitions from any mode to this mode. This is the same as **WLM_ACTIVE**, but only CPU resources are regulated. Other resources (memory, disk IO, and total limits when enabled) are still accounted.

WLM_PASSIVE
Turns WLM on in **passive** mode or transitions from any mode to **passive** mode.

WLM_BIND_RSETS
Requests that WLM takes the resource set bindings into account.

WLM_PROCTOTAL
Enables process total limits on resource usage.

WLM_CLASSTOTAL
Enables class total limits on resource usage.

Some combinations of the flags above are not legal:

- **WLM_OFF**, **WLM_ACTIVE**, **WLM_CPU_ONLY**, and **WLM_PASSIVE** are mutually exclusive.

- **WLM_BIND_RSETS**, **WLM_PROCTOTAL**, and **WLM_CLASSTOTAL**, are ineffective when used together with **WLM_OFF**.
- Only **WLM_TEST_ON** is allowed to non-root users.
- If **WLM_TEST_ON** is specified, the other flags are ineffective and should not be specified.

Return Values

Upon successful completion, the **wlm_set** subroutine returns a value of 0, and the current state of WLM is returned in the *flags* parameter. The return value is **WLM_OFF**, **WLM_ACTIVE**, **WLM_CPU_ONLY**, or **WLM_PASSIVE**. When WLM is on in either mode, the **WLM_BIND_RSETS**, **WLM_PROCTOTAL**, and **WLM_CLASSTOTAL**, flags are added when appropriate.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

Related Information

The **wlmcntrl** command.

The **wlm.h** header file.

The **wlm_load** (“wlm_load Subroutine” on page 2348) subroutine.

wlm_set_tag Subroutine

Purpose

Sets the current process's tag and related flags.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

#include <sys/user.h>

int wlm_set_tag ( tag,  flags)

char *tag;

int *flags;
```

Description

The **tag** attribute is an attribute of a process that can be set using the Workload Manager (WLM) **wlm_set_tag** subroutine. This tag is a character string with a maximum length of **WLM_TAG_LENGTH** (not including the null terminator). Process tags can be displayed using the **ps** command.

The **tag** attribute is also one of the **process** attributes used in the assignment rules to automatically assign a process to a given class. The syntax of the assignment rules precludes the use of special characters in the application tag string. Thus, application tags should be comprised only of upper and lower case letters, numbers and underscores ('_').

The main use of the **tag** attribute is to allow WLM administrators to discriminate between several instances of the same application, which typically have the same user and group ids, execute the same binary, and, therefore, end up in the same class using the standard classification criteria.

For more details about application tags, refer to Workload Manager application programming interface in *Operating system and device management*.

When an application sets its tag using **wlm_set_tag**, it is automatically reclassified according to the current assignment rules and the new tag is taken into account when doing this reclassification.

In addition to the tag itself, the application can also specify flags indicating to WLM if a child process should inherit the tag from its parent after a **fork** or an **exec** subroutine.

A process does not require any special privileges to set its tag.

Parameters

Item	Description
tag	Specifies the address of a character string. An error is returned if this tag is too long.
flags	Specifies the address of an integer interpreted in a manner similar to the **versflags** field of the **wlmargs** structure passed to other API routines. The integer pointed to by flags should be initialized with **WLM_VERSION**. In addition, one or more of the following values can be or'ed to **WLM_VERSION**: **SWLMTAGINHERITFORK** Specifies that the children of this process inherit the parent's tag on the **fork** subroutine. **SWLMTAGINHERITEXEC** Specifies that the process retains its tag after a call to the **exec** subroutine. Both flags can be set to specify that the children of a tagged process inherits the tag on the **fork** subroutine and then retains it on the **exec** subroutine.

Return Values

Upon successful completion, the **wlm_set_tag** subroutine returns a value of 0. In case of error, a non-0 value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

wlm_set_thread_tag Subroutine

Purpose

Sets the current thread's tag and related flags.

Library

Workload Manager Library (**libwlm.a**)

Syntax

```
#include <sys/wlm.h>

int wlm_set_thread_tag ( *tag, *flags)
```

Description

The `wlm_set_thread_tag` subroutine sets or unsets the tag on the current thread. The tag is a character string with a maximum length of the value set with the `WLM_TAG_LENGH` macro (not including the null terminator). The tag on the thread can be unset by passing a NULL value for the *tag* parameter or by passing a pointer to a NULL tag.

Setting the tag attribute at the thread-level assigns a thread-level class to the current thread. This allows discriminating between different threads of the same process or application, whereas standard classification criteria fails due to the following reasons:

- These threads have the same user and group IDs (unless the threads have per-thread credentials).
- These threads run the same binary.
- These threads have the same process-level tag.

For a thread with a thread-level tag attribute, the thread-level tag, fixed priority, status, and credentials are used in place of those belonging to the application to classify the thread. The thread-level class is independent and unrelated to the process-level class and is also determined based on the rules of the current WLM configuration.

In addition to the tag itself, the thread also specifies flags indicating to WLM the tag inheritance policy on a **fork**, **exec** or **pthread_create** subroutine.

Thread tags can be displayed using the `ps` command. A thread does not require any special privileges to set its tag.

This subroutine is only supported when running in 1:1 mode and will fail if it is invoked by a thread belonging to a process that is running in M:N mode. Threads are only regulated by WLM if their scheduling policy is set to SCHED_OTHER.

Parameters

Item	Description
tag	Specifies the address of a character string. An error is returned if the length of this tag exceeds the value set by the `WLM_TAG_LENGH` macro.

Item	Description
flags	Specifies the address of an integer interpreted in a manner similar to the **versflags** field of the **wlmargs** structure passed to other API routines. The integer that flags pointed to should be initialized with the **WLM_VERSION** macro. In addition, a bitwise OR operation can be applied on the **WLM_VERSION** macro and one or more of the following values: **TWLMTAGINHERITFORK** Specifies that if the tagged thread makes a `fork` system call, the child process will inherit the parent's tag. The thread-level tag and class will become process-based in the child. **TWLMTAGINHERITEXEC** Specifies that if the tagged thread makes an `exec` system call, the process will inherit the parent's tag. The thread-level tag and class will become process based in the process that calls the `exec` subroutine. The process will inherit the thread-level class if class inheritance is ON for the class or if it was manually assigned; otherwise it will be reclassified according to WLM rules.

Return Values

Upon successful completion, the **wlm_set_thread_tag** subroutine returns a value of 0. In case of error, a non-0 value is returned.

Error Codes

For a list of the possible error codes returned by the WLM API functions, see the description of the **wlm.h** header file.

Implementation Specifics

This subroutine is part of the Base Operating System (BOS) Runtime.

wmemchr Subroutine

Purpose

Find a wide-character in memory.

Library

Standard library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
wchar_t *wmemchr (const wchar_t * ws, wchar_t wc, size_t n) ;
```

Description

The **wmemchr** function locates the first occurrence of **wc** in the initial **n** wide-characters of the object pointed to be **ws**. This function is not affected by locale and all **wchar_t** values are treated identically. The null wide-character and **wchar_t** values not corresponding to valid characters are not treated specially.

If **n** is zero, **ws** must be a valid pointer and the function behaves as if no valid occurrence of **wc** is found.

Return Values

The **wmemchr** function returns a pointer to the located wide-character, or a null pointer if the wide-character does not occur in the object.

wmemcmp Subroutine

Purpose

Compare wide-characters in memory.

Library

Standard library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
int wmemcmp (const wchar_t * ws1, const wchar_t * ws2, size_t n);
```

Description

The **wmemcmp** function compares the first **n** wide-characters of the object pointed to by **ws1** to the first n wide-characters of the object pointed to by **ws2**. This function is not affected by locale and all **wchar_t** values are treated identically. The null wide-character and **wchar_t** values not corresponding to valid characters are not treated specially.

If **n** is zero, **ws1** and **ws2** must be a valid pointers and the function behaves as if the two objects compare equal.

Return Values

The **wmemcmp** function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by **ws1** is greater than, equal to, or less than the object pointed to by **ws2**.

wmemcpy Subroutine

Purpose

Copy wide-characters in memory.

Library

Standard library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
wchar_t *wmemcpy (wchar_t * ws1, const wchar_t * ws2, size_t n) ;
```

Description

The **wmemcpy** function copies **n** wide-characters from the object pointed to by **ws2** to the object pointed to be **ws1**. This function is not affected by locale and all **wchar_t** values are treated identically. The null wide-character and **wchar_t** values not corresponding to valid characters are not treated specially.

If **n** is zero, **ws1** and **ws2** must be a valid pointers, and the function copies zero wide-characters.

Return Values

The **wmemcpy** function returns the value of **ws1**.

wmemmove Subroutine

Purpose

Copy wide-characters in memory with overlapping areas.

Library

Standard library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
wchar_t *wmemmove (wchar_t * ws1, const wchar_t * ws2, size_t n) ;
```

Description

The **wmemmove** function copies **n** wide-characters from the object pointed to by **ws2** to the object pointed to by **ws1**. Copying takes place as if the **n** wide-characters from the object pointed to by **ws2** are first copied into a temporary array of **n** wide-characters that does not overlap the objects pointed to by **ws1** or **ws2**, and then the **n** wide-characters from the temporary array are copied into the object pointed to by ws1.

This function is not affected by locale and all **wchar_t** values are treated identically. The null wide-character and **wchar_t** values not corresponding to valid characters are not treated specially.

If **n** is zero, **ws1** and **ws2** must be a valid pointers, and the function copies zero wide-characters.

Return Values

The **wmemmove** function returns the value of **ws1**.

wmemset Subroutine

Purpose

Set wide-characters in memory.

Library

Standard library (**libc.a**)

Syntax

```
#include <wchar.h>
```

```
wchar_t *wmemset (wchar_t * ws, wchar_t wc, size_t n);
```

Description

The **wmemset** function copies the value of wc into each of the first n wide-characters of the object pointed to by ws. This function is not affected by locale and all wchar_t values are treated identically. The null wide-character and wchar_t values not corresponding to valid characters are not treated specially. If n is zero, ws must be a valid pointer and the function copies zero wide-characters.

Return Values

The **wmemset** functions returns the value of ws.

wordexp Subroutine

Purpose

Expands tokens from a stream of words.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wordexp.h>
```

```
int wordexp ( Words,  Pwordexp,  Flags)
const char *Words;
wordexp_t *Pwordexp;
int Flags;
```

Description

The **wordexp** subroutine performs word expansions equivalent to the word expansion that would be performed by the shell if the contents of the *Words* parameter were arguments on the command line. The list of expanded words are placed in the *Pwordexp* parameter. The expansions are the same as that which would be performed by the shell if the *Words* parameter were the part of a command line representing the parameters to a command. Therefore, the *Words* parameter cannot contain an unquoted <newline> character or any of the unquoted shell special characters | (pipe), & (ampersand), ; (semicolon), < (less than sign), or > (greater than sign), except in the case of command substitution. The *Words* parameter also cannot contain unquoted parentheses or braces, except in the case of command or variable substitution. If the *Words* parameter contains an unquoted comment character # (number sign) that is the beginning of a token, the **wordexp** subroutine may treat the comment character as a regular character, or may interpret it as a comment indicator and ignore the remainder of the expression in the *Words* parameter.

The **wordexp** subroutine allows an application to perform all of the shell's expansions on a word or words obtained from a user. For example, if the application prompts for a file name (or a list of file names) and

then uses the **wordexp** subroutine to process the input, the user could respond with anything that would be valid as input to the shell.

The **wordexp** subroutine stores the number of generated words and a pointer to a list of pointers to words in the *Pwordexp* parameter. Each individual field created during the field splitting or path name expansion is a separate word in the list specified by the *Pwordexp* parameter. The first pointer after the last last token in the list is a null pointer. The expansion of special parameters `*` (asterisk), `@` (at sign), `#` (number sign), `?` (question mark), `-` (minus sign), $ (dollar sign), `!` (exclamation point), and `0` is unspecified.

The words are expanded in the order shown below:

1. Tilde expansion is performed first.
2. Parameter expansion, command substitution, and arithmetic expansion are performed next, from beginning to end.
3. Field splitting is then performed on fields generated by step 2, unless the IFS (input field separators) is full.
4. Path-name expansion is performed, unless the **set -f** command is in effect.
5. Quote removal is always performed last.

Parameters

Item	Description
Flags	Contains a bit flag specifying the configurable aspects of the **wordexp** subroutine.
Pwordexp	Contains a pointer to a **wordexp_t** structure.
Words	Specifies the string containing the tokens to be expanded.

The value of the *Flags* parameter is the bitwise, inclusive OR of the constants below, which are defined in the **wordexp.h** file.

Item	Description
WRDE_APPEND	Appends words generated to those generated by a previous call to the **wordexp** subroutine.
WRDE_DOOFFS	Makes use of the **we_offs** structure. If the **WRDE_DOOFFS** flag is set, the *we_offs* structure is used to specify the number of null pointers to add to the beginning of the **we_words** structure. If the **WRDE_DOOFFS** flag is not set in the first call to the **wordexp** subroutine with the *Pwordexp* parameter, it should not be set in subsequent calls to the **wordexp** subroutine with the *Pwordexp* parameter.
WRDE_NOCMD	Fails if command substitution is requested.
WRDE_REUSE	The *Pwordexp* parameter was passed to a previous successful call to the **wordexp** subroutine. Therefore, the memory previously allocated may be reused.
WRDE_SHOWERR	Does not redirect standard error to **/dev/null**.
WRDE_UNDEF	Reports error on an attempt to expand an undefined shell variable.

The **WRDE_ APPEND** flag can be used to append a new set of words to those generated by a previous call to the **wordexp** subroutine. The following rules apply when two or more calls to the **wordexp** subroutine are made with the same value of the *Pwordexp* parameter and without intervening calls to the **wordfree** subroutine:

1. The first such call does not set the **WRDE_ APPEND** flag. All subsequent calls set it.
2. For a single invocation of the **wordexp** subroutine, all calls either set the **WRDE_DOOFFS** flag, or do not set it.
3. After the second and each subsequent call, the *Pwordexp* parameter points to a list containing the following:

a. Zero or more null characters, as specified by the **WRDE_DOOFFS** flag and the **we_offs** structure.
b. Pointers to the words that were in the *Pwordexp* parameter before the call, in the same order as before.
c. Pointers to the new words generated by the latest call, in the specified order.

4. The count returned in the *Pwordexp* parameter is the total number of words from all of the calls.
5. The application should not modify the *Pwordexp* parameter between the calls.

The **WRDE_NOCMD** flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell commands. Disallowing unquoted shell special characters also prevents unwanted side effects such as executing a command or writing to a file.

Unless the **WRDE_SHOWERR** flag is set in the *Flags* parameter, the **wordexp** subroutine redirects standard error to the **/dev/null** file for any utilities executed as a result of command substitution while expanding the *Words* parameter. If the **WRDE_SHOWERR** flag is set, the **wordexp** subroutine may write messages to standard error if syntax errors are detected while expanding the *Words* parameter.

The *Pwordexp* structure is allocated by the caller, but memory to contain the expanded tokens is allocated by the **wordexp** subroutine and added to the structure as needed.

The *Words* parameter cannot contain any `<newline>` characters, or any of the unquoted shell special characters `|`, &, `;`, `()`, `{}`, <, or >, except in the context of command substitution.

Return Values

If no errors are encountered while expanding the *Words* parameter, the **wordexp** subroutine returns a value of 0. If an error occurs, it returns a nonzero value indicating the error.

Errors

If the **wordexp** subroutine terminates due to an error, it returns one of the nonzero constants below, which are defined in the **wordexp.h** file.

Item	Description
WRDE_BADCHAR	One of the unquoted characters `|`, &, `;`, <, >, parenthesis, or braces appears in the *Words* parameter in an inappropriate context.
WRDE_BADVAL	Reference to undefined shell variable when the **WRDE_UNDEF** flag is set in the *Flags* parameter.
WRDE_CMDSUB	Command substitution requested when the **WRDE_NOCMD** flag is set in the *Flags* parameter.
WRDE_NOSPACE	Attempt to allocate memory was unsuccessful.
WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.

If the **wordexp** subroutine returns the error value **WRDE_SPACE**, then the expression in the *Pwordexp* parameter is updated to reflect any words that were successfully expanded. In other cases, the *Pwordexp* parameter is not modified.

wordfree Subroutine

Purpose

Frees all memory associated with the *Pwordexp* parameter.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wordexp.h>
```

void wordfree (*Pwordexp*)
wordexp_t *Pwordexp;

Description

The **wordfree** subroutine frees any memory associated with the *Pwordexp* parameter from a previous call to the **wordexp** subroutine.

Parameters

Item	Description
Pwordexp	Structure containing a list of expanded words.

wpar_getcid Subroutine

Purpose

Returns the configured workload partition (WPAR) identifier for the current process.

Library

Standard C Library (`libc.a`)

Syntax

```
#include <sys/wpar.h>
cid_t wpar_getcid (void)
```

Description

The **wpar_getcid** subroutine returns the configured identifier associated with the workload partition of the current process. If the current process is executing within the global environment, **wpar_getcid** subroutine returns the value of zero. If the current process is executing within a workload partition, the workload partition subroutine returns a nonzero value. This identifier can be different each time that a workload partition is started on a system.

Return Values

The `wpar_getcid` subroutine returns the following values:

Item	Description
0	The process is executing within the global environment.
nonzero	Configured workload partition identification number.

wpar_getckey Subroutine

Purpose

Returns the static workload partition identifier for the current process.

Library

Standard C Library (`libc.a`)

Syntax

```
#include <sys/wpar.h>
ckey_t wpar_getckey (void)
```

Description

The **wpar_getckey** subroutine returns the workload partition static identifier that is associated with the current process. If the current process is executing within the global environment, the **wpar_getckey** subroutine returns a value of zero. If the current process is executing within a workload partition, the **wpar_getckey** subroutine returns a value of nonzero. This identifier that the **wpar_getckey** subroutine returns is the same each time when the workload partition starts, unless that partition is removed from that system.

Return Values

Item	Description
`0`	Process is executing within the global environment.
`nonzero`	Static workload partition identification number.

wpar_log_err Subroutine

Purpose

Logs an error message for a specific WPAR.

Library

libwparlog.a

Syntax

```
#include <wpars/wparlog.h>
```

```
int wpar_log_err(
kcid, cat_file_name,
msg_set_no, msg_no,
default_fmt_msg, ...)
cid_t kcid;
char * cat_file_name;
unsigned int msg_set_no;
unsigned int msg_no;
char * default_fmt_msg;
```

Description

The **wpar_log_err** interface provides a mechanism to log error messages for a given WPAR. Each WPAR can hold up to 1 KB of error message. If there is enough space to log the new message, the command logs the message otherwise it fails. When called from a process inside the WPAR, the *kcid* parameter should match the **CID** of that WPAR. Otherwise the routine will report failure.

Parameters

Item	Description
kcid	**CID** of the WPAR. The **CID** can be obtained from the WPAR name using the **getcorralid** and **corral_getcid** system calls.
cat_file_name	Catalog file name to be used for translation
msg_set_no	Message sets the number of the error messages in the catalog file
msg_no	Message number of the error message
default_fmt_msg	<Need description>
...	Arguments to the message if any

Return Values

Item	Description
0	Successful completion
-1	Failure

Error codes

Item	Description
ENOMEM	Not enough memory
EPERM	No permission to log message into the specified WPAR
EINVAL	Invalid parameter

Example

```
/*Log a error message into WPAR with cid 4.*/
…
wpar_log_err(4, “wpararrs.cat”,1,10,”%s : command failed”, “mycommand”);
…
```

wpar_print_err Subroutine

Purpose

Writes error messages of a specific WPAR into a file.

Library

libwparlog.a

Syntax

```
#include <wpars/wparlog.h>
```

```
int wpar_print_err( kcid, file)
cid_t kcid;
FILE * file;
```

Description

The **wpar_print_err** interface writes all the error messages of a WPAR logged using the **wparerr**, **wpar_err**, and **kwpar_err** into the given file. The file should be opened in write or append mode. The interface cannot be called from inside WPAR.

Parameters

Item	Description
kcid	CID of the WPAR. The CID can be obtained from the WPAR name using the **getcorralid** system call.
file	File that stores the error messages.

Return Values

Item	Description
0	Successful completion
-1	Failure

Error codes

Item	Description
ENOMEM	Not enough memory
EPERM	No permission to log message into the specified WPAR
EINVAL	Invalid parameter

Example

```
/*To write messages of WPAR with cid 4 into stderr.
*/
wpar_print_err(4, stderr);
```

write, writex, write64x, writev, writevx, ewrite, ewritev, pwrite, or pwritev Subroutine

Purpose

Writes to a file.

Library

Item	Description
write, **writex**, **write64x**, **writev**, **writevx**, **pwrite**, **pwritev**	Standard C Library (**libc.a**)
ewrite, **ewritev**	MLS Library (**libmls.a**)

Syntax

```
#include <unistd.h>
```

```
ssize_t write (FileDescriptor, Buffer, NBytes)
int  FileDescriptor;
const void * Buffer;
size_t  NBytes;

int writex (FileDescriptor, Buffer, NBytes, Extension)
int FileDescriptor;
char *Buffer;
unsigned int NBytes;
int  Extension;

int write64x (FileDescriptor, Buffer, NBytes, Extension)
int FileDescriptor;
void *Buffer;
size_t NBytes;
void *Extension;

ssize_t pwrite (FileDescriptor, Buffer, NBytes, Offset)
int  FileDescriptor;
const void * Buffer;
size_t  NBytes;
off_t Offset;
```

```
#include <sys/uio.h>
```

```
ssize_t writev (FileDescriptor, iov, iovCount)
int FileDescriptor;
const struct iovec *  iov;
int  iovCount;

ssize_t writevx (FileDescriptor, iov, iovCount, Extension)
int FileDescriptor;
struct iovec *iov;
int iovCount;
int  Extension;
```

```
#include <unistd.h>
#include <sys/uio.h>
```

```
ssize_t pwritev (
int  FileDescriptor,
const struct iovec * iov,
int iovCount,
offset_t offset);

ssize_t ewrite (FileDescriptor, Buffer, Nbytes,  labels)
int  FileDescriptor;
const void * Buffer;
size_t  NBytes;
sec_labels_t * labels;

ssize_t ewritev (FileDescriptor, iov, iovCount,labels)
int FileDescriptor;
const struct iovec * iov;
int  iovCount;
sec_labels_t * labels;
```

Description

The **write** subroutine attempts to write the number of bytes of data that is specified by the *NBytes* parameter to the file associated with the *FileDescriptor* parameter from the buffer pointed to by the *Buffer* parameter.

The **writev** subroutine runs the same action but gathers the output data from the *iovCount* buffers specified by the array of **iovec** structures pointed to by the *iov* parameter. Each **iovec** entry specifies the base address and length of an area in memory from which data is written. The **writev** subroutine always writes a complete area before it proceeds to the next.

The **writex** and **writevx** subroutines are the same as the **write** and **writev** subroutines, with the addition of an *Extension* parameter, which is used to write to some device drivers.

With regular files and devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from the **write** subroutine, the file pointer increments by the number of bytes written.

With devices incapable of seeking, writing always begins at the current position. The value of a file pointer that is associated with such a device is undefined.

If a **write** requests that more bytes be written than there is room for (for example, the **ulimit** or the physical end of a medium), only as many bytes as there is room for are written. For example, suppose there is space for 20 bytes more in a file before it reaches a limit. A **write** of 512 bytes returns 20. The next write of a non-zero number of bytes gives a failure return (except as noted in current topic) and the implementation generates a **SIGXFSZ** signal for the thread.

Fewer bytes can be written than requested if there is not enough room to satisfy the request. Here, the number of bytes written is returned. The next attempt to write a nonzero number of bytes is unsuccessful (except as noted in the following text). The limit that is reached can be either that set by the **ulimit** subroutine or the end of the physical medium.

Successful completion of a **write** subroutine clears the **SetUserID** bit (**S_ISUID**) of a file if all of the following are true:

- The calling process does not have root user authority.
- The effective user ID of the calling process does not match the user ID of the file.
- The file is executable by the group (**S_IXGRP**) or other (**S_IXOTH**).

The **write** subroutine clears the **SetGroupID** bit (**S_ISGID**) if all of the following are true:

- The calling process does not have root user authority.
- The group ID of the file does not match the effective group ID or one of the supplementary group IDs of the process.
- The file is executable by the owner (**S_IXUSR**) or others (**S_IXOTH**).

Note: Clearing of the **SetUserID** and **SetGroupID** bits can occur even if the **write** subroutine is unsuccessful, if file data was modified before the error was detected.

If the **O_APPEND** flag of the file status is set, the file offset is set to the end of the file before each write.

If the *FileDescriptor* parameter refers to a regular file whose file status flags specify **O_SYNC**, this action is a synchronous update (as described in the **open** subroutine).

If the *FileDescriptor* parameter refers to a regular file that a process opens with the **O_DEFER** file status flag set, the data and file size are not updated on permanent storage until a process issues an **fsync** subroutine or conducts a synchronous update. If all processes that opened the file with the **O_DEFER** file status flag set close the file before a process issues an **fsync** subroutine or conducts a synchronous update, the data and file size are not updated on permanent storage.

Write requests to a pipe (or first-in-first-out (FIFO)) are handled the same as a regular file with the following exceptions:

- There is no file offset associated with a pipe; hence, each write request appends to the end of the pipe.
- If the size of the write request is less than or equal to the value of the **PIPE_BUF** system variable (described in the **pathconf** routine), the **write** subroutine is automatic. The data is not interleaved with data from other write processes on the same pipe. Writes of greater than **PIPE_BUF** bytes can have data that is interleaved, on arbitrary boundaries, with writes by other processes, whether the **O_NDELAY** or **O_NONBLOCK** file status flags are set.
- If the **O_NDELAY** and **O_NONBLOCK** file status flags are clear (the default), a write request to a full pipe causes the process to block until enough space becomes available to handle the entire request.
- If the **O_NDELAY** file status flag is set, a write to a full pipe returns a 0.
- If the **O_NONBLOCK** file status flag is set, a write to a full pipe returns a value of **-1** and sets the **errno** global variable to **EAGAIN**.

When the systems attempts to write to a character special file that supports nonblocking writes and no data can currently be written (streams are an exception that described later):

- If the **O_NDELAY** and **O_NONBLOCK** flags are clear (the default), the **write** subroutine blocks until data can be written.
- If the **O_NDELAY** flag is set, the **write** subroutine returns 0.
- If the **O_NONBLOCK** flag is set, the **write** subroutine returns **-1** and sets the **errno** global variable to **EAGAIN** if no data can be written.

When the systems attempts to write to a regular file that supports enforcement-mode record locks, and all or part of the region to be written is locked by another process, the following can occur:

- If the **O_NDELAY** and **O_NONBLOCK** file status flags are clear (the default), the calling process blocks until the lock is released.
- If the **O_NDELAY** or **O_NONBLOCK** file status flag is set, then the **write** subroutine returns a value of **-1** and sets the **errno** global variable to **EAGAIN**.

Note: The **fcntl** subroutine provides more information about record locks.

If *fildes* refers to a STREAM, the operation of **write** is determined by the values of the minimum and maximum *nbyte* range ("packet size") accepted by the STREAM. These values are determined by the topmost STREAM module. If *nbyte* falls within the packet size range, *nbyte* bytes are written. If *nbyte* does not fall within the range and the minimum packet size value is 0, **write** breaks the buffer into maximum packet size segments before it sends the data downstream (the last segment contains less than the maximum packet size). If *nbyte* does not fall within the range and the minimum value is non-zero, **write** fails with **errno** set to **ERANGE**. Writing a zero-length buffer (*nbyte* is 0) to a STREAMS device sends 0 bytes with 0 returned. However, writing a zero-length buffer to a STREAMS-based pipe or FIFO sends no message and **0** is returned. The process can issue **I_SWROPT ioctl** to enable zero-length messages to be sent across the pipe or FIFO.

When the system writes to a STREAM, data messages are created with a priority band of 0. When it is writing to a STREAM that is not a pipe or FIFO:

- **O_NONBLOCK** specify either **O_NONBLOCK** or **O_NDELAY**. The IBM streams implementation treats these two the same.
- If **O_NONBLOCK** or **O_NDELAY** is clear, and the STREAM cannot accept data (the STREAM write queue is full because of internal flow control conditions), **write** blocks until data can be accepted.
- If **O_NONBLOCK** or **O_NDELAY** is set and the STREAM cannot accept data, **write** returns **-1** and set **errno** to **EAGAIN**.
- If **O_NONBLOCK** or **O_NDELAY** is set and part of the buffer was written while a condition in which the STREAM cannot accept more data occurs, **write** ends and return the number of bytes written.

Note: The IBM streams implementation treats **O_NONBLOCK** and **O_NDELAY** the same.

In addition, **write** and **writev** fail if the STREAM head processes an asynchronous error before the call. Here, the value of **errno** does not reflect the result of **write** or **writev** but reflects the prior error.

The **writev** function is equivalent to **write**, but gathers the output data from the **iovcnt** buffers specified by the members of the **iov array: iov[0], iov[1], ..., iov[iovcnt - 1]**. **iovcnt** is valid if greater than 0 and less than or equal to **{IOV_MAX}**, defined in **limits.h**.

Each **iovec** entry specifies the base address and length of an area in memory from which data is written. The **writev** function always writes a complete area before it proceeds to the next.

If *fildes* refers to a regular file and all of the **iov_len** members in the array pointed to by **iov** are **0**, **writev** returns **0** and have no other effect. For other file types, the behavior is unspecified.

If the sum of the **iov_len** values is greater than **SSIZE_MAX**, the operation fails and no data is transferred.

The behavior of an interrupted **write** subroutine depends on how the handler for the arriving signal was installed. The handler can be installed in one of two ways, with the following results:

- If the handler is installed with an indication that subroutines not be restarted, the **write** subroutine returns a value of **-1** and sets the **errno** global variable to **EINTR** (even if some data was already written).
- If the handler is installed with an indication that subroutines be restarted, and:
  - If no data was written when the interrupt was handled, the **write** subroutine does not return a value (it is restarted).
  - If data was written when the interrupt was handled, this **write** subroutine returns the amount of data already written.

Note: A write to a regular file is not interruptible. Only the writes to objects that can block indefinitely, such as FIFOs, sockets, and some devices, are interruptible. If *fildes* refers to a socket, **write** is equivalent to the **send** subroutine with no flags set.

The **write64x** subroutine is the same as the **writex** subroutine, where the *Extension* parameter is a pointer to a **j2_ext** structure (see the **j2/j2_cntl.h** file). The **write64x** subroutine is used to write an encrypted file in raw mode (see **O_RAW** in the **fcntl.h** file). Using the **O_RAW** flag on encrypted files has the same limitations as using **O_DIRECT** on regular files.

The **ewrite** and **ewritev** subroutines write to a stream and set the security attributes. The **ewrite** subroutine copies the number of bytes of the data that is specified by the *Nbyte* parameter from the buffer pointed to by the *Buffer* parameter to a stream associated with the *FileDescriptor* parameter. Security information for the message is set to the values in the structure pointed to by the *labels* parameter.

The **pwrite** function conducts the same action as **write**, except that it writes into a given position without changing the file pointer. The first three arguments to **pwrite** are the same as **write** with the addition of a fourth argument that is offset for the wanted position inside the file.

```
ssize_t pwrite64(int fd , const void *buf , size_t nbytes , off64_t offset)
```

The **pwrite64** subroutine conducts the same action as **pwrite** but the limit of offset to the maximum file size for the file that is associated with the **fileDescriptor** and **DEV_OFF_MAX** if the file associated with **fileDescriptor** is a block special or character special file.

Using the **write** or **pwrite** subroutine with a file descriptor obtained from a call to the **shm_open** subroutine fails with **ENXIO**.

The **pwritev** subroutine conducts the same action as the **writev** subroutine, except that the **pwritev** subroutine writes to the given position in the file without changing the file pointer. The first three arguments of the **pwritev** subroutine are the same as the **writev** subroutine with the addition of the *offset* argument that points to the position that you want inside the file. An error occurs when the file that the **pwritev** subroutine writes to is incapable of seeking.

Parameters

Item	Description
Buffer	Identifies the buffer that contains the data to be written.
Extension	Provides communication with character device drivers that require more information or return additional status. Each driver interprets the *Extension* parameter in a device-dependent way, either as a value or as a pointer to a communication area. Drivers must apply reasonable defaults when the *Extension* parameter value is 0.
FileDescriptor	Identifies the object to which the data is to be written.

Item	Description

iov	Points to an array of **iovec** structures, which identifies the buffers that contain the data to be written. The **iovec** structure is defined in the **sys/uio.h** file and contains the following members:
	`caddr_t  iov_base;`
	`size_t  iov_len;`

iovCount	Specifies the number of **iovec** structures pointed to by the *iov* parameter.
NBytes	Specifies the number of bytes to write.
offset	The position in the file where the writing begins.
labels	A pointer to the extended security attribute structure.

Return Values

Upon successful completion, the **write**, **writex**, **write64x**, **writev**, **writevx**, and **pwritev** subroutines return the number of bytes that were written. The number of bytes written is never greater than the value specified by the *NBytes* parameter. Otherwise, a value of **-1** is returned and the **errno** global variable is set to indicate the error.

Upon successful completion, the **ewrite** and **ewritev** subroutines return a value of 0. Otherwise, the global variable **errno** is set to identify the error.

Error Codes

The **write**, **writex**, **write64x**, **writev**, **writevx**, **ewrite**, **ewritev**, and **pwritev** subroutines are unsuccessful when one or more of the following are true:

Item	Description
EAGAIN	The **O_NONBLOCK** flag is set on this file and the process would be delayed in the write operation; or an enforcement-mode record lock is outstanding in the portion of the file that is to be written.
EBADF	The *FileDescriptor* parameter does not specify a valid file descriptor open for writing.
EDQUOT	New disk blocks cannot be allocated for the file because the user or group quota of disk blocks is exhausted on the file system.
EFBIG	An offset greater than **MAX_FILESIZE** was requested on the 32-bit kernel.
EFAULT	The *Buffer* parameter or part of the *iov* parameter points to a location outside of the allocated address space of the process.
EFBIG	An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. If the user sets the environment variable **XPG_SUS_ENV=ON** before execution of the process, then the **SIGXFSZ** signal is posted to the process when it exceeds the process' file size limit.
EINVAL	The file position pointer that is associated with the *FileDescriptor* parameter was negative; the *iovCount* parameter value was not 1 - 16, inclusive; or one of the **iov_len** values in the **iov** array was negative.
EINVAL	The sum of the **iov_len** values from a 32-bit application overflowed a 32-bit signed integer in either a 32-bit or a 64-bit kernel environment, or the sum of the **iov_len** values from a 64-bit application overflowed a 32-bit signed integer in a 32-bit kernel environment.
EINVAL	The STREAM or multiplexer that is referenced by**fileDescriptor** is linked (directly or indirectly) downstream from a multiplexer.

Item	Description
EINVAL	The value of the *Nbytes* parameter that is larger than **OFF_MAX**, was requested on the 32-bit kernel. Here, the system call is requested from a 64-bit application that is running on a 32-bit kernel.
EINTR	A signal was caught during the write operation, and the signal handler was installed with an indication that subroutines are not to be restarted.
EIO	An I/O error occurred while the system is writing to the file system; or the process is a member of a background process group that is attempting to write to its control terminal, **TOSTOP** is set, the process is not ignoring or blocking **SIGTTOU**, and the process group has no parent process.
ENOSPC	No free space is left on the file system that contains the file.
ENXIO	A hangup occurred on the STREAM being written to. The **write** or **pwrite** subroutine was used with a file descriptor obtained from a call to the **shm_open** subroutine.
EPIPE	An attempt was made to write to a file that is not opened for reading by any process, or to a socket of type **SOCK_STREAM** that is not connected to a peer socket; or an attempt was made to write to a pipe or FIFO that is not open for reading by any process. If this condition occurs, a **SIGPIPE** signal is sent to the process.
ERANGE	The transfer request size was outside the range that is supported by the STREAMS file that is associated with *FileDescriptor*.

The **write**, **writex**, **writev**, **writevx**, and **pwritev** subroutines might be unsuccessful if the following is true:

Item	Description
ENXIO	A request was made of a nonexistent device, or the request was outside the capabilities of the device.
EFBIG	An attempt was made to write to a regular file where *NBytes* greater than zero and the starting offset is greater than or equal to the offset maximum established in the open file description that is associated with *FileDescriptor*.
EINVAL	The **offset** argument is invalid. The value is negative.
ESPIPE	*fildes* is associated with a pipe or FIFO.

The **write64x** subroutine was unsuccessful if the **EINVAL** error code is returned:

Item	Description
EINVAL	The **j2_ext** structure was not initialized correctly. For example, the version was wrong, or the file was not encrypted.
EINVAL	The **j2_ext** structure is passed by using the **J2EXTCMD_RDRAW** command for files that were not opened in raw-mode.

The **ewrite** and **ewritev** subroutines were unsuccessful if one of the following error codes is true:

Item	Description
ENOMEM	The memory or space is too small.
EACCES	Permission is denied. The user does not have sufficient privilege to write data.
ERESTART	**ERESTART** is used to determine whether a system call is restartable.

wstring Subroutine

Purpose

Perform operations on wide character strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wstring.h>

wchar_t *wstrcat (“wstring Subroutine” on page 2372) (XString1, XString2)
wchar_t *XString1, *XString2;

wchar_t * wstrncat (XString, XString2, Number)
wchar_t *XString1, *XString2;
int Number;

int  wstrcmp (XString1, XString2)
wchar_t *XString1, *XString2;

int  wstrncmp (XString1, XString2, Number)
wchar_t *XString1, *XString2;
int Number;

wchar_t * wstrcpy (XString1, XString2)
wchar_t *XString1, *XString2;

wchar_t * wstrncpy (XString1, XString2, Number)
wchar_t *XString1, *XString2;
int Number;

int  wstrlen (XString)
wchar_t *XString;

wchar_t * wstrchr (XString, Number)
wchar_t *XString;
int Number;

wchar_t * wstrrchr (XString, Number)
wchar_t *XString;
int Number;

wchar_t * wstrpbrk (XString1, XString2)
wchar_t *XString1, XString2;

int  wstrspn (XString1, XString2)
wchar_t *XString1, XString2;

int  wstrcspn (XString1, XString2)
wchar_t *XString1, XString2;

wchar_t * wstrtok (XString1, XString2)
wchar_t *XString1, XString2;
```

```
wchar_t * wstrdup (XString1)
wchar_t *XString1;
```

Description

The **wstring** subroutines copy, compare, and append strings in memory, and determine location, size, and existence of strings in memory. For these subroutines, a string is an array of **wchar_t** characters, terminated by a null character. The **wstring** subroutines parallel the **string** subroutines, but operate on strings of type **wchar_t** rather than on type **char**, except as specifically noted below.

The parameters *XString1*, *XString2*, and *XString* point to strings of type **wchar_t** (arrays of **wchar** characters terminated by a **wchar_t** null character).

The subroutines **wstrcat**, **wstrncat**, **wstrcpy**, and **wstrncpy** all alter the *XString1* parameter. They do not check for overflow of the array pointed to by *XString1*. All string movement is performed wide character by wide character. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **wstring.h** file.

The **wstrcat** subroutine appends a copy of the **wchar_t** string pointed to by the *XString2* parameter to the end of the **wchar_t** string pointed to by the *XString1* parameter. The **wstrcat** subroutine returns a pointer to the null-terminated result.

The **wstrncat** subroutine copies, at most, the value of the *Number* parameter of **wchar_ t** characters in the *XString2* parameter to the end of the **wchar_t** string pointed to by the *XString1* parameter. Copying stops before *Number* **wchar_t** character if a null character is encountered in the string pointed to by the *XString2* parameter. The **wstrncat** subroutine returns a pointer to the null-terminated result.

The **wstrcmp** subroutine lexicographically compares the **wchar_t** string pointed to by the *XString1* parameter to the **wchar_t** string pointed to by the *XString2* parameter. The **wstrcmp** subroutine returns a value that is:

- Less than 0 if *XString1* is less than *XString2*
- Equal to 0 if *XString1* is equal to *XString2*
- Greater than 0 if *XString1* is greater than *XString2*

The **wstrncmp** subroutine makes the same comparison as **wstrcmp**, but it compares, at most, the value of the *Number* parameter of pairs of **wchar** characters. The comparisons are based on collation values as determined by the locale category **LC_COLLATE** and the **LANG** variable.

The **wstrcpy** subroutine copies the string pointed to by the *XString2* parameter to the array pointed to by the *XString1* parameter. Copying stops when the **wchar_t** null is copied. The **wstrcpy** subroutine returns the value of the *XString1* parameter.

The **wstrncpy** subroutine copies the value of the *Number* parameter of **wchar_t** characters from the string pointed to by the *XString2* parameter to the **wchar_t** array pointed to by the *XString1* parameter. If *XString2* is less than *Number* **wchar_t** characters long, then **wstrncpy** pads *XString1* with trailing null characters to fill *Number* **wchar_t** characters. If *XString2* is *Number* or more **wchar_t** characters long, only the first *Number* **wchar_t** characters are copied; the result is not terminated with a null character. The **wstrncpy** subroutine returns the value of the *XString1* parameter.

The **wstrlen** subroutine returns the number of **wchar_t** characters in the string pointed to by the *XString* parameter, not including the terminating **wchar_t** null.

The **wstrchr** subroutine returns a pointer to the first occurrence of the **wchar_t** specified by the *Number* parameter in the **wchar_t** string pointed to by the *XString* parameter. A null pointer is returned if the **wchar_t** does not occur in the **wchar_t** string. The **wchar_t** null that terminates a string is considered to be part of the **wchar_t** string.

The **wstrrchr** subroutine returns a pointer to the last occurrence of the character specified by the *Number* parameter in the **wchar_t** string pointed to by the *XString* parameter. A null pointer is returned if the **wchar_t** does not occur in the **wchar_t** string. The **wchar_t** null that terminates a string is considered to be part of the **wchar_t** string.

The **wstrpbrk** subroutine returns a pointer to the first occurrence in the **wchar_t** string pointed to by the *XString1* parameter of any code point from the string pointed to by the *XString2* parameter. A null pointer is returned if no character matches.

The **wstrspn** subroutine returns the length of the initial segment of the string pointed to by the *XString1* parameter that consists entirely of code points from the **wchar_t** string pointed to by the *XString2* parameter.

The **wstrcspn** subroutine returns the length of the initial segment of the **wchar_t** string pointed to by the *XString1* parameter that consists entirely of code points *not* from the **wchar_t** string pointed to by the *XString2* parameter.

The **wstrtok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the *XString1* parameter. The *XString2* parameter specifies a set of code points as token delimiters. If the *XString1* parameter is anything other than null, then the **wstrtok** subroutine reads the string pointed to by the *XString1* parameter until it finds one of the delimiter code points specified by the *XString2* parameter. It then stores a **wchar_t** null into the **wchar_t** string, replacing the delimiter code point, and returns a pointer to the first **wchar_t** of the text token. The **wstrtok** subroutine keeps track of its position in the **wchar_t** string so that subsequent calls with a null *XString1* parameter step through the **wchar_t** string. The delimiters specified by the *XString2* parameter can be changed for subsequent calls to **wstrtok**. When no tokens remain in the **wchar_t** string pointed to by the *XString1* parameter, the **wstrtok** subroutine returns a null pointer.

The **wstrdup** subroutine returns a pointer to a **wchar_t** string that is a duplicate of the **wchar_t** string to which the *XString1* parameter points. Space for the new string is allocated using the **malloc** subroutine. When a new string cannot be created, a null pointer is returned.

wstrtod or watof Subroutine

Purpose

Converts a string to a double-precision floating-point.

Library

Standard C Library

Syntax

```
#include <wstring.h>
```

double wstrtod (*String*, *Pointer*)
wchar_t *String, **Pointer;

```
double watof (String)
wchar_t *String;
```

Description

The **wstrtod** subroutine returns a double-precision floating-point number that is converted from an **wchar_t** string pointed to by the *String* parameter. The system searches the *String* until it finds the first unrecognized character.

The **wstrtod** subroutine recognizes a string that starts with any number of white-space characters (defined by the **iswspace** subroutine), followed by an optional sign, a string of decimal digits that may include a decimal point, e or E, an optional sign or space, and an integer.

When the value of *Pointer* is not **(wchar_t **)** null, a pointer to the search terminating character is returned to the address indicated by *Pointer*. When the resulting number cannot be created, **Pointer* is set to *String* and 0 (zero) is returned.

The **watof (***String***)** subroutine functions like the **wstrtod (***String* **(wchar_t **)** null**)**.

Parameters

Item	Description
String	Specifies the address of the string to scan.
Pointer	Specifies the address at which the pointer to the terminating character is stored.

Error Codes

When the value causes overflow, **HUGE_VAL** (defined in the **math.h** file) is returned with the appropriate sign, and the **errno** global variable is set to **ERANGE**. When the value causes underflow, 0 is returned and the **errno** global variable is set to **ERANGE**.

wstrtol, watol, or watoi Subroutine

Purpose

Converts a string to an integer.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <wstring.h>
```

```
long wstrtol ( String,  Pointer,  Base)
wchar_t *String, **Pointer;
int Base;
```

```
long watol (String)
wchar_t *String;
```

```
int watoi (String)
wchar_t *String;
```

Description

The **wstrtol** subroutine returns a long integer that is converted from the string pointed to by the *String* parameter. The string is searched until a character is found that is inconsistent with *Base*. Leading white-space characters defined by the **ctype** subroutine **iswspace** are ignored.

When the value of *Pointer* is not **(wchar_t **)** null, a pointer to the terminating character is returned to the address indicated by *Pointer*. When an integer cannot be created, the address indicated by *Pointer* is set to *String*, and 0 is returned.

When the value of *Base* is positive and not greater than 36, that value is used as the base during conversion. Leading zeros that follow an optional leading sign are ignored. When the value of *Base* is 16, 0x and 0X are ignored.

When the value of *Base* is 0, the system chooses an appropriate base after examining the actual string. An optional sign followed by a leading zero signifies octal, and a leading 0x or 0X signifies hexadecimal. In all other cases, the subroutines assume a decimal base.

Truncation from **long** data type to **int** data type occurs by assignment, and also by explicit casting.

The **watol (***String***)** subroutine functions like **wstrtol (***String*, **(wchar_t **)** null**, 10)**.

The **watoi (***String***)** subroutine functions like **(int) wstrtol (***String*, **(wchar_t **)** null**, 10)**.

Note: Even if overflow occurs, it is ignored.

Parameters

Item	Description
String	Specifies the address of the string to scan.
Pointer	Specifies the address at which the pointer to the terminating character is stored.
Base	Specifies an integer value used as the base during conversion.

x

The following Base Operating System (BOS) runtime services begin with the letter *x*.

xcrypt_key_setup, xcrypt_encrypt, xcrypt_decrypt, xcrypt_hash, xcrypt_malloc, xcrypt_free, xcrypt_printb, xcrypt_mac, xcrypt_hmac, xcrypt_sign, xcrypt_verify, xcrypt_dh_keygen, xcrypt_dh, xcrypt_btoa and xcrypt_randbuff Subroutine

Purpose

Provides various block and stream cipher algorithms and two crypto-secure hash algorithms.

Library

Cryptographic Library (**libmodcrypt.a**)

Syntax

```
#include <xcrypt.h>

int xcrypt_key_setup (alg, key, keymat, keysize, dir)
int alg;
xcrypt_key *key;
u_char *keymat;
int keysize;
int dir;
```

```
int xcrypt_encrypt (alg, mode, key, IV, in, insize, out, padding)
int alg;
int mode;
xcrypt_key *key;
u_char *IV;
u_char *in;
int insize;
u_char *out;
int padding;
```

```
int xcrypt_decrypt (alg, mode, key, IV, in, insize, out, padding)
int alg;
int mode;
xcrypt_key *key;
u_char *IV;
u_char *in;
int insize;
u_char *out;
int padding;
```

```
int xcrypt_hash (alg, in, insize, out)
int alg;
u_char *in;
int insize;
u_char *out;
```

```
int xcrypt_malloc (pp, size, blocksize)
uchar **pp;
int size;
int blocksize;
```

```
void xcrypt_free (p, size)
void *p;
int size;
```

```
void xcrypt_printb (p, size)
void *p;
int size;
```

```
int xcrypt_mac (alg, key, in, insize, mac)
int alg;
xcrypt_key *key;
u_char *in;
int insize;
u_char *mac;
```

```
int xcrypt_hmac (alg, key, in, insize, out)
int alg;
xcrypt_key *key;
u_char *in;
int insize;
u_char *out;
```

```
int xcrypt_sign (alg, key, in, insize, sig)
int alg;
xcrypt_key *key;
u_char *in;
int insize;
u_char *sig;
```

```
int xcrypt_verify (alg, key, in, insize, sig, sigsize)
int alg;
xcrypt_key *key;
u_char *in;
int insize;
u_char *sig;
int sigsize;
```

```
int xcrypt_dh_keygen (dh_pk, keysize)
void **dh_pk;
int keysize;
```

```
int xcrypt_dh (dh_pk, in, out)
void dh_pk;
u_char *in;u_char *out;
```

```
void xcrypt_btoa (dest, buff, size)
char *dest;
void *buff;
int size;
```

```
void xcrypt_randbuff (dest, size)
void *dest;
int size;
```

Description

These subroutines provide block and stream cipher algorithms, plus two crypto-secure hash algorithms. Encryption may be done through the Rijndael, Mars, and Twofish block ciphers or the SEAL stream cipher. Each of these algorithms uses a use a block length of 128 bits and key lengths of 128, 192 and 256 bits. SEAL is a stream cipher that uses a 160 bit key and a 32 bit word input stream. In addition, the MD5 and SHA-1 cryptographic hash algorithms are included.

The `libmodcrypt.a` library and associated headers are available through the Expansion Pack.

The **xcrypt_key_setup** subroutine is used to setup a key schedule for any of the block cipher algorithms. It stores the key schedule in the **xcrypt_key** data structure that is passed in. If key scheduling is done for HMAC, set the *dir* parameter of `xcrypt_key_setup` to NULL. Note that when using the Twofish

method, the *keymat* parameter should also be set to NULL. The `xcrypt_key_setup` subroutine expects the *keymat* pointer to point to a PKCS#8 object when used with public key encryption. Then the *keysize* parameter indicates the size of the PCKS#8 object, not the size of the key.

The **xcrypt_encrypt** subroutine encrypts a buffer. Data can be encrypted using the CBC mode (Cipher Block Chaining), EBC mode (Electronic Codebook) or CBF1 mode. Note that when EBC mode is being used, no initalization vector is required.

The **xcrypt_decrypt** subroutine decrypts a buffer. Data can be encrypted using the CBC mode (Cipher Block Chaining), EBC mode (Electronic Codebook) or CBF1 mode. If the `xcrypt_encrypt` subroutine is called with padding on, the `xcrypt_decrypt` subroutine must also be called with padding on. It is the caller's responsibility to determine whether padding is used. Note that when EBC mode is being used, no initalization vector is required.

The **xcrypt_hash** subroutine hashes a buffer using either the MD5 or SHA-1 algorithm.

The **xcrypt_malloc** subroutine dynamically allocates the least size bytes of memory to provide blocks of *blocksize* bytes. For example, if *size* is 105 and *blocksize* is 10, the **xcrypt_malloc** subroutine will return at least 110 bytes of memory (11 blocks, each 10 bytes in size). The **xcrypt_malloc** subroutine should be used when you need **xcrypt** to pad buffers. It will make sure that enough memory is allocated for the data to be encrypted, plus the padding.

The **xcrypt_free** subroutine overwrites and frees dynamically allocated memory.

The **xcrypt_printb** subroutine prints a buffer to the screen in hexadecimal notation.

The **xcrypt_mac** subroutine provides the caller with a Message Authentication Code (MAC). DES is the only algorithm that is supported.

The **xcrypt_hmac** subroutine provides the caller with a Hashed Message Authentication Code (HMAC). The algorithm used is MD5 or SHA-1.

The **xcrypt_sign** subroutine allows the caller to sign data using public key mechanisms.

The **xcrypt_verify** subroutine allows the caller to verify private key signatures.

The **xcrypt_dh_keygen** subroutine returns the private key object to be used in Diffie Helman key agreement. The *dh* parameter should point to NULL. The *keysize* parameter can be KEY_512, KEY_1024, or KEY_2048.

The **xcrypt_dh** subroutine allows the caller to execute the two steps to compute a shared secret through the Diffie-Hellman key agreement algorithm. If *in* is NULL, then *out* contains the public shared object to be transmitted to the other party involved in the key agreement. Otherwise, *in* points to the public shared object received from another party, and *out* points to the shared private key.

The **xcrypt_btoa** subroutine returns a string representing the buffer in hexadecimal. Note that the *dest* parameter must point to a buffer of *size* * 2 + 1.

The **xcrypt_randbuff** subroutine fills a buffer with random data.

Parameters

Item	Description
alg	Specifies the cipher to use. You can use the following symbolic constants: **RIJNDAEL** Rijndael (AES) block cipher. Supports key sizes of 128, 192, and 256 bits. **MARS** Mars block cipher. Supports key sizes of 128, 192, and 256 bits. **TWOFISH** Twofish block cipher. Supports key sizes of 128, 192, and 256 bits. **SEAL** SEAL stream cipher. Supports key sizes of 128, 192, and 256 bits. **SHA1** SHA-1 one-way hash function. Arbitrary lengths are permitted. **MD5** MD5 one-way hash function. Arbitrary lengths are permitted. **DES** Data Encryption Standard. Supports key sizes of 64 bits. **TDES** Triple Data Encryption Standard. Supports key sizes of 64 and 128 bits. **MAC_DES** Message Authentication Code using the DES algorithm. Supports key sizes of 64 bits. **CAST5** CAST encryption algorithm. Supports key sizes of 40, 80, and 128 bits. **RSA** Rivest, Shamir Adleman. The *keysize* passed to `xcrypt_key_setup` should be the size of the PKCS#8 object. **DSA** Digital Signature Algorithm. The *keysize* passed to `xcrypt_key_setup` should be the size of the PKCS#8 object.
key	Points to the key instance to set up. Use for encryption or decryption.
keymat	Points to the key material used to build the key schedule.
keysize	Size of the *keymat* parameter. You can use the following symbolic constants: **KEY_64** 64 bit key **KEY_80** 80 bit key **KEY_128** 128 bit key **KEY_192** 192 bit key **KEY_256** 256 bit key

Item	Description
dir	The direction (encryption or decryption). You can use the following symbolic constants: **DIR_ENCRYPT** Encrypt **DIR_DECRYPT** Decrypt
mode	Specifies the mode of operation. You can use the following symbolic constants: **MODE_ECB** Ciphering in ECB mode **MODE_CBC** Ciphering in CBC mode **MODE_CFB1** Ciphering in 1-bit CFB mode
IV	Points to the buffer holding the initialization vector. **Note:** When using ECB mode, the *IV* parameter should point to NULL.
in	Points to the buffer holding the data to encrypt, decrypt, or hash.
insize	Contains the size of the *in* parameter.
mac	Points to an output buffer that will hold the MAC.
out	Points to a preallocated output buffer.
padding	Specifies whether **xcrypt** should pad the buffers or not. You can use the following symbolic constants: **TRUE** True **FALSE** False
pp	A double pointer to the destination.
sig	Points to an output buffer that holds the RSA signature.
sigsize	The size of *sig* in bytes.
size	Contains the amount of memory to allocate, deallocate, print the contents of, or convert to a string.

Item	Description
blocksize	Contains the size of the blocks. You can use the following symbolic constants: **BITS_32** 32 bits **BITS_80** 80 bits **BITS_128** 128 bits **BITS_160** 160 bits **BITS_192** 192 bits **BITS_256** 256 bits **DES_BLOCKSIZE** 64 bits
p	Points to the memory to overwrite and free.
buff	Points to a buffer to print or convert to a string.
dest	Points to a preallocated destination buffer.
dh_pk	Refers to the private key object to be passed to `xcrypt_dh`. The private key object is obtained by calling `xcrypt_dh_keygen` before calling `xcryp`.

Return Values

The **xcrypt_key_setup**, **xcrypt_hash** and `xcrypt_dh_keygen` subroutines return 0 on success. The **xcrypt_malloc** subroutine returns the amount of memory allocated on success. The **xcrypt_encrypt** subroutine returns the amount of data encrypted on success. The **xcrypt_decrypt** subroutine returns the amount of data decrypted on success.

Upon success, the `xcrypt_mac` subroutine returns the size of *mac* in bytes; the `xcrypt_hmac` subroutine returns the size of hashed *mac* in bytes; the `xcrypt_sig` subroutine returns the size of signature; and the `xcrypt_dh` subroutine returns the number of bytes written to *out*. The `xcrypt_verify` subroutine returns a value of 1 to indicate successful signal verification.

On failure the previous subroutines return the following error codes:

Error Codes

xcrypt_key_setup:

Item	Description
BAD_ALIGN32	A parameter is not aligned on a 32 bit boundary.
BAD_KEY_DIR	The *dir* parameter is not valid
BAD_KEY_INSTANCE	The *key* parameter is not valid
BAD_KEY_MAT	The *keysize* parameter is not valid or the *key* parameter is corrupt.

xcrypt_encrypt:

Item	Description
BAD_ALG	The *alg* parameter is not valid.

Item	Description
BAD_CIPHER_MODE	The *mode* parameter is not valid.
BAD_CIPHER_STATE	The *key* parameter is not valid.
BAD_INPUT_LEN	The *insize* parameter is not a multiple of of the *blocksize* being used by a block cipher for encryption or decryption.
BAD_IV	The *IV* parameter is set to NULL when the *mode* parameter is set to **MODE_CBC**.
BAD_IV_MAT	The *IV* parameter is not valid.
BAD_KEY_INSTANCE	The *key* parameter is not valid.

xcrypt_decrypt:

Item	Description
BAD_ALG	The *alg* parameter is not valid.
BAD_CIPHER_MODE	The *mode* parameter is not valid.
BAD_CIPHER_STATE	The *key* parameter is not valid.
BAD_INPUT_LEN	The *insize* parameter is not a multiple of of the *blocksize* being used by a block cipher for encryption or decryption.
BAD_IV	The *IV* parameter is set to NULL when the *mode* parameter is set to **MODE_CBC**.
BAD_IV_MAT	The *IV* parameter is not valid.
BAD_KEY_INSTANCE	The *key* parameter is not valid.

xcrypt_hash:

Item	Description
BAD_ALG	The *alg* parameter is not valid.

xcrypt_malloc:

Item	Description
BAD_MEM_ALLOC	The system could not allocate *size* bytes.

y

The following Base Operating System (BOS) runtime services begin with the letter *y*.

yield Subroutine

Purpose

Yields the processor to processes with higher priorities.

Library

Standard C library (**libc.a**)

Syntax

```
void yield (void);
```

Description

The **yield** subroutine forces the current running process or thread to relinquish use of the processor. If the run queue is empty when the **yield** subroutine is called, the calling process or kernel thread is immediately rescheduled. If the calling process has multiple threads, only the calling thread is affected. The process or thread resumes execution after all threads of equal or greater priority are scheduled to run.

Notices

This information was developed for products and services offered in the US.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year).

Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

Privacy policy considerations

IBM Software products, including software as a service solutions, (“Software Offerings”) may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as the customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y