AIX Version 7.1

Technical Reference: Communication Subroutines

Note

Before using this information and the product it supports, read the information in “Notices” on page 445 .

This edition applies to AIX Version 7.1 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright International Business Machines Corporation 2015, 2018.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this document..... vii
Highlighting..... vii
Case-sensitivity in AIX..... vii
ISO 9000..... vii

Communication subroutines..... 1
eXternal Data Representation..... 1
xdr_accepted_reply Subroutine..... 1
xdr_array Subroutine..... 2
xdr_bool Subroutine..... 2
xdr_bytes Subroutine..... 3
xdr_callhdr Subroutine..... 4
xdr_callmsg Subroutine..... 5
xdr_char Subroutine..... 5
xdr_destroy Macro..... 6
xdr_enum Subroutine..... 7
xdr_float Subroutine..... 7
xdr_free Subroutine..... 8
xdr_getpos Macro..... 9
xdr_hyper Subroutine..... 10
xdr_inline Macro..... 10
xdr_int Subroutine..... 11
xdr_long Subroutine..... 12
xdr_opaque Subroutine..... 12
xdr_opaque_auth Subroutine..... 13
xdr_pmap Subroutine..... 14
xdr_pmaplist Subroutine..... 15
xdr_pointer Subroutine..... 15
xdr_reference Subroutine..... 16
xdr_rejected_reply Subroutine..... 17
xdr_replymsg Subroutine..... 18
xdr_setpos Macro..... 18
xdr_short Subroutine..... 19
xdr_string Subroutine..... 20
xdr_u_char Subroutine..... 21
xdr_u_int Subroutine..... 21
xdr_u_long Subroutine..... 22
xdr_u_short Subroutine..... 23
xdr_union Subroutine..... 24
xdr_vector Subroutine..... 25
xdr_void Subroutine..... 25
xdr_wrapstring Subroutine..... 26
xdr_authunix_parms Subroutine..... 27
xdr_double Subroutine..... 27
xdrmem_create Subroutine..... 28
xdrrec_create Subroutine..... 29
xdrrec_endofrecord Subroutine..... 30
xdrrec_eof Subroutine..... 31
xdrrec_skiprecord Subroutine..... 31
xdrstdio_create Subroutine..... 32

Network Information Services ... 33
yp_all Subroutine ... 33
yp_bind Subroutine ... 35
yp_first Subroutine ... 36
yp_get_default_domain Subroutine ... 37
yp_master Subroutine ... 37
yp_match Subroutine ... 38
yp_next Subroutine ... 39
yp_order Subroutine ... 41
yp_unbind Subroutine ... 41
yp_update Subroutine ... 42
yperr_string Subroutine ... 43
ypprot_err Subroutine ... 44
Network Information Services+ ... 45
nis_add_entry (NIS+ API) ... 45
nis_first_entry (NIS+ API) ... 48
nis_list (NIS+ API) ... 52
nis_local_directory (NIS+ API) ... 56
nis_lookup (NIS+ API) ... 57
nis_modify_entry (NIS+ API) ... 60
nis_next_entry (NIS+ API) ... 63
nis_perror (NIS+ API) ... 67
nis_remove_entry (NIS+ API) ... 67
nis_sperror (NIS+ API) ... 71
Simple Network Management Protocol (SNMP) ... 71
getsmuxEntrybyname or getsmuxEntrybyidentity Subroutine ... 72
isodetailor Subroutine ... 73
ll_hdinit, ll_dbinit, _ll_log, or ll_log Subroutine ... 74
o_number, o_integer, o_string, o_igeneric, o_generic, o_specific, or o_ipaddr Subroutine ... 76
oid_cmp, oid_cpy, oid_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode_aux, prim2oid, or oid2prim Subroutine ... 79
oid_extend or oid_normalize Subroutine ... 81
readobjects Subroutine ... 82
s_generic Subroutine ... 83
smux_close Subroutine ... 84
smux_error Subroutine ... 85
smux_free_tree Subroutine ... 85
smux_init Subroutine ... 86
smux_register Subroutine ... 87
smux_response Subroutine ... 89
smux_simple_open Subroutine ... 90
smux_trap Subroutine ... 91
smux_wait Subroutine ... 92
text2inst, name2inst, next2inst, or nextot2inst Subroutine ... 93
text2oid or text2obj Subroutine ... 95
Sockets ... 95
_ ... 96
a ... 99
b ... 104
c ... 107
d ... 111
e ... 113
f ... 131
g ... 144
h ... 191
i ... 194
kvalid_user Subroutine ... 234
listen Subroutine ... 235

n.... 236
PostQueuedCompletionStatus Subroutine.... 238
r.... 240
s.... 281
WriteFile Subroutine.... 350
Packet Capture.... 352
ioctl BPF Control Operations.... 352
Librdmacm Library.... 354
Returned error rules.... 354
Supported verbs.... 355
Device Management.... 387
Memory region management.... 388
Libibverbs Library.... 391
Returned error rules.... 391
Supported Verbs.... 391
Verbs not supported by the libibverbs library.... 414
select Subroutine Interface for Data Link Control (DLC) Devices 442

Notices.... 445
Privacy policy considerations.... 446
Trademarks.... 447

Index.... 449

About this document

This topic collection provides experienced C programmers with complete detailed information about eXternal Data Representation, Network Information Services (NIS), and Simple Network Management Protocol (SNMP), sockets, packet capture, Librdman library, and Libibverbs library for the AIX® operating system. To use the topic collection effectively, you should be familiar with commands, system calls, subroutines, file formats, and special files. This publication is also available on the documentation CD that is shipped with the operating system.

Highlighting

The following highlighting conventions are used in this document:

Item	Description
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.

© Copyright IBM Corp. 2015, 2018

Technical Reference: Communication subroutines

Review the communication subroutines for eXternal data representation, Network Information Services (NIS), NIS+, Simple Network Management Protocol (SNMP), and sockets.

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).

eXternal Data Representation

This topic collection includes the subroutines that help in external data representation in the required format.

xdr_accepted_reply Subroutine

Purpose

Encodes RPC reply messages.

Library

C Library (**libc.a**)

Syntax

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

int xdr_accepted_reply ( xdrs,  ar)
XDR *xdrs;
struct accepted_reply *ar;
```

Description

The **xdr_accepted_reply** subroutine encodes Remote Procedure Call (RPC) reply messages. The routine generates message replies similar to RPC message replies without using the RPC program.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ar	Specifies the address of the structure that contains the RPC reply.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information

List of RPC Programming References
eXternal Data Representation (XDR) Overview for Programming
Remote Procedure Call (RPC) Overview for Programming

© Copyright IBM Corp. 2015, 2018

xdr_array Subroutine

Purpose

Translates between variable-length arrays and their corresponding external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>

xdr_array (xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR * xdrs;
char ** arrp;
u_int * sizep;
u_int  maxsize;
u_int  elsize;
xdrproc_t  elproc;
```

Description

The **xdr_array** subroutine is a filter primitive that translates between variable-length arrays and their corresponding external representations. This subroutine is called to encode or decode each element of the array.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
arrp	Specifies the address of the pointer to the array. If the *arrp* parameter is null when the array is being deserialized, the XDR program allocates an array of the appropriate size and sets the parameter to that array.
sizep	Specifies the address of the element count of the array. The element count cannot exceed the value for the *maxsize* parameter.
maxsize	Specifies the maximum number of array elements.
elsize	Specifies the byte size of each of the array elements.
elproc	Translates between the C form of the array elements and their external representations. This parameter is an XDR filter.

Related information

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

xdr_bool Subroutine

Purpose

Translates between Booleans and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdr_bool ( xdrs,  bp)
XDR *xdrs;
bool_t *bp;
```

Description

The **xdr_bool** subroutine is a filter primitive that translates between Booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
bp	Specifies the address of the Boolean data.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information

List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_bytes Subroutine

Purpose

Translates between internal counted byte arrays and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdr_bytes ( xdrs,  sp,  sizep,  maxsize)
XDR *xdrs;
char **sp;
u_int *sizep;
u_int maxsize;
```

Description

The **xdr_bytes** subroutine is a filter primitive that translates between counted byte arrays and their external representations. This subroutine treats a subset of generic arrays, in which the size of array

elements is known to be 1 and the external description of each element is built-in. The length of the byte array is explicitly located in an unsigned integer. The byte sequence is not terminated by a null character. The external representation of the bytes is the same as their internal representation.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
sp	Specifies the address of the pointer to the byte array.
sizep	Points to the length of the byte area. The value of this parameter cannot exceed the value of the *maxsize* parameter.
maxsize	Specifies the maximum number of bytes allowed when XDR encodes or decodes messages.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information

List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_callhdr Subroutine

Purpose

Describes RPC call header messages.

Library

C Library (**libc.a**)

Syntax

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

```
xdr_callhdr ( xdrs,  chdr)
XDR *xdrs;
struct rpc_msg *chdr;
```

Description

The **xdr_callhdr** subroutine describes Remote Procedure Call (RPC) call header messages. This subroutine generates call headers that are similar to RPC call headers without using the RPC program.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
chdr	Points to the structure that contains the header for the call message.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of RPC Programming References
eXternal Data Representation (XDR) Overview for Programming
Remote Procedure Call (RPC) Overview for Programming

xdr_callmsg Subroutine

Purpose

Describes RPC call messages.

Library

C Library (**libc.a**)

Syntax

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

xdr_callmsg ( xdrs,  cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;
```

Description

The **xdr_callmsg** subroutine describes Remote Procedure Call (RPC) call messages. This subroutine generates messages similar to RPC messages without using the RPC program.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
cmsg	Points to the structure that contains the text of the call message.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of RPC Programming References
eXternal Data Representation (XDR) Overview for Programming
Remote Procedure Call (RPC) Overview for Programming

xdr_char Subroutine

Purpose

Translates between C language characters and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdr_char (*xdrs*, *cp*)
XDR ***xdrs*;
char ***cp*;

Description

The **xdr_char** subroutine is a filter primitive that translates between C language characters and their external representations.

Note: Encoded characters are not packed and occupy 4 bytes each. For arrays of characters, the programmer should consider using the **xdr_bytes**, **xdr_opaque**, or **xdr_string** routine.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
cp	Points to the character.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_destroy Macro

Purpose

Destroys the XDR stream pointed to by the *xdrs* parameter.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

void xdr_destroy (*xdrs*)
XDR ***xdrs*;

Description

The **xdr_destroy** macro invokes the destroy routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter and frees the private data structures allocated to the stream. The use of the XDR stream handle is undefined after it is destroyed.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

xdr_enum Subroutine

Purpose

Translates between a C language enumeration (enum) and its external representation.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdr_enum (*xdrs*, *ep*)
XDR ***xdrs*;
enum_t ***ep*;

Description

The **xdr_enum** subroutine is a filter primitive that translates between a C language enumeration (enum) and its external representation.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ep	Specifies the address of the enumeration data.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_float Subroutine

Purpose

Translates between C language floats and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdr_float (*xdrs*, *fp*)
XDR ***xdrs*;
float ***fp*;

Description

The **xdr_float** subroutine is a filter primitive that translates between C language floats (normalized single-precision floating-point numbers) and their external representations.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
fp	Specifies the address of the float.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_free Subroutine

Purpose

Deallocates, or frees, memory.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

void xdr_free (*proc*, *objp*)
xdrproc_t *proc*;
char ***objp*;

Description

The **xdr_free** subroutine is a generic freeing routine that deallocates memory. The *proc* parameter specifies the eXternal Data Representation (XDR) routine for the object being freed. The *objp* parameter is a pointer to the object itself.

Note: The pointer passed to this routine is *not* freed, but the object it points to *is* freed (recursively).

Parameters

Item	Description
proc	Points to the XDR stream handle.
objp	Points to the object being freed.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

xdr_getpos Macro

Purpose

Returns an unsigned integer that describes the current position in the data stream.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

u_int xdr_getpos (*xdrs*)
XDR ****xdrs*;

Description

The **xdr_getpos** macro invokes the get-position routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. This routine returns an unsigned integer that describes the current position in the data stream.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.

Return Values

This macro returns an unsigned integer describing the current position in the stream. In some XDR streams, it returns a value of -1, even though the value has no meaning.

Related reference
xdr_setpos Macro
Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

xdr_hyper Subroutine

Purpose

Translates long integers from C language to their external representations.

Library

C Library (**libc.a**)

Syntax

```
int xdr_hyper(XDR *xdrs, long long *lp)
```

Description

A filter primitive that translates ANSI C long integers to their external representations. This subroutine returns 1 if it succeeds, otherwise returns a value of 0.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ulp	Specifies the address of the long integer.

Return Values

Upon successful completion, the `xdr_hyper` subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

xdr_inline Macro

Purpose

Returns a pointer to the buffer of a stream pointed to by the *xdrs* parameter.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
long *x_inline ( xdrs,  len)
XDR *xdrs;
int len;
```

Description

The **xdr_inline** macro invokes the inline subroutine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The subroutine returns a pointer to a contiguous piece of the stream's buffer, whose size is specified by the *len* parameter. The buffer can be used for any purpose, but it is not data-portable. The **xdr_inline** macro may return a value of null if it cannot return a buffer segment of the requested size.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.
len	Specifies the size, in bytes, of the internal buffer.

Return Values

This macro returns a pointer to a piece of the stream's buffer.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

xdr_int Subroutine

Purpose

Translates between C language integers and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>

xdr_int ( xdrs,  ip)
XDR *xdrs;
int *ip;
```

Description

The **xdr_int** subroutine is a filter primitive that translates between C language integers and their external representations.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ip	Specifies the address of the integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_long Subroutine

Purpose

Translates between C language long integers and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdr_long
( xdrs,  lp)
XDR *xdrs;
long *lp;
```

Description

The **xdr_long** filter primitive translates between C language long integers and their external representations. This primitive is characteristic of most eXternal Data Representation (XDR) library primitives and all client XDR routines.

Parameters

Item	Description
xdrs	Points to the XDR stream handle. This parameter can be treated as an opaque handler and passed to the primitive routines.
lp	Specifies the address of the number.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

When in 64 BIT mode, if the value of the long integer can not be expressed in 32 BIT, **xdr_long** will return a value of 0.

Related information

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

xdr_opaque Subroutine

Purpose

Translates between fixed-size opaque data and its external representation.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdr_opaque (*xdrs*, *cp*, *cnt*)
XDR ***xdrs**;
char ***cp**;
u_int *cnt*;

Description

The **xdr_opaque** subroutine is a filter primitive that translates between fixed-size opaque data and its external representation.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
cp	Specifies the address of the opaque object.
cnt	Specifies the size, in bytes, of the object. By definition, the actual data contained in the opaque object is not machine-portable.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_opaque_auth Subroutine

Purpose

Describes RPC authentication messages.

Library

C Library (**libc.a**)

Syntax

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

xdr_opaque_auth (*xdrs*, *ap*)
XDR ***xdrs**;
struct opaque_auth ***ap**;

Description

The **xdr_opaque_auth** subroutine describes Remote Procedure Call (RPC) authentication information messages. It generates RPC authentication message data without using the RPC program.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ap	Points to the structure that contains the authentication information.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of RPC Programming References
eXternal Data Representation (XDR) Overview for Programming
Remote Procedure Call (RPC) Overview for Programming

xdr_pmap Subroutine

Purpose

Describes parameters for **portmap** procedures.

Library

C Library (**libc.a**)

Syntax

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

xdr_pmap ( xdrs,  regs)
XDR *xdrs;
struct pmap *regs;
```

Description

The **xdr_pmap** subroutine describes parameters for **portmap** procedures. This subroutine generates **portmap** parameters without using the **portmap** interface.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
regs	Points to the buffer or register where the **portmap** daemon stores information.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
portmap subroutine
List of RPC Programming References
eXternal Data Representation (XDR) Overview for Programming

xdr_pmaplist Subroutine

Purpose

Describes a list of port mappings externally.

Library

C Library (**libc.a**)

Syntax

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

```
xdr_pmaplist ( xdrs,  rp)
XDR *xdrs;
struct pmaplist **rp;
```

Description

The **xdr_pmaplist** subroutine describes a list of port mappings externally. This subroutine generates the port mappings to Remote Procedure Call (RPC) ports without using the **portmap** interface.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
rp	Points to the structure that contains the **portmap** listings.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information

portmap subroutine

List of RPC Programming References

eXternal Data Representation (XDR) Overview for Programming

xdr_pointer Subroutine

Purpose

Provides pointer chasing within structures and serializes null pointers.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdr_pointer (xdrs, objpp, objsize, xdrobj)
XDR * xdrs;
char ** objpp;
```

```
u_int  objsize;
xdrproc_t  xdrobj;
```

Description

The **xdr_pointer** subroutine provides pointer chasing within structures and serializes null pointers. This subroutine can represent recursive data structures, such as binary trees or linked lists.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
objpp	Points to the character pointer of the data structure.
objsize	Specifies the size of the structure.
xdrobj	Specifies the XDR filter for the object.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

xdr_reference Subroutine

Purpose

Provides pointer chasing within structures.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdr_reference ( xdrs,  pp,  size,  proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

Description

The **xdr_reference** subroutine is a filter primitive that provides pointer chasing within structures. This primitive allows the serializing, deserializing, and freeing of any pointers within one structure that are referenced by another structure.

The **xdr_reference** subroutine does not attach special meaning to a null pointer during serialization. Attempting to pass the address of a null pointer can cause a memory error. The programmer must describe data with a two-armed discriminated union. One arm is used when the pointer is valid; the other arm, when the pointer is null.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
pp	Specifies the address of the pointer to the structure. When decoding data, XDR allocates storage if the pointer is null.
size	Specifies the byte size of the structure pointed to by the *pp* parameter.
proc	Translates the structure between its C form and its external representation. This parameter is the XDR procedure that describes the structure.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

xdr_rejected_reply Subroutine

Purpose

Describes RPC message rejection replies.

Library

C Library (**libc.a**)

Syntax

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

xdr_rejected_reply ( xdrs,  rr)
XDR *xdrs;
struct rejected_reply *rr;
```

Description

The **xdr_rejected_reply** subroutine describes Remote Procedure Call (RPC) message rejection replies. This subroutine can be used to generate rejection replies similar to RPC rejection replies without using the RPC program.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
rr	Points to the structure that contains the rejected reply.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of RPC Programming References
eXternal Data Representation (XDR) Overview for Programming
Remote Procedure Call (RPC) Overview for Programming

xdr_replymsg Subroutine

Purpose

Describes RPC message replies.

Library

C Library (**libc.a**)

Syntax

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

xdr_replymsg ( xdrs,  rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

Description

The **xdr_replymsg** subroutine describes Remote Procedure Call (RPC) message replies. Use this subroutine to generate message replies similar to RPC message replies without using the RPC program.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
rmsg	Points to the structure containing the parameters of the reply message.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of RPC Programming References
eXternal Data Representation (XDR) Overview for Programming
Remote Procedure Call (RPC) Overview for Programming

xdr_setpos Macro

Purpose

Changes the current position in the XDR stream.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdr_setpos (*xdrs*, *pos*)
XDR ***xdrs**;
u_int *pos*;

Description

The **xdr_setpos** macro invokes the set-position routine associated with the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The new position setting is obtained from the **xdr_getpos** macro. The **xdr_setpos** macro returns a value of false if the set position is not valid or if the requested position is out of bounds.

A position cannot be set in some XDR streams. Trying to set a position in such streams causes the macro to fail. This macro also fails if the programmer requests a position that is not in the stream's boundaries.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.
pos	Specifies a position value obtained from the **xdr_getpos** macro.

Return Values

Upon successful completion (if the stream is positioned successfully), this macro returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference
xdr_getpos Macro
Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming

xdr_short Subroutine

Purpose

Translates between C language short integers and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
xdr_short ( xdrs, sp)
XDR *xdrs;
short *sp;
```

Description

The **xdr_short** subroutine is a filter primitive that translates between C language short integers and their external representations.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
sp	Specifies the address of the short integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_string Subroutine

Purpose

Translates between C language strings and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>

xdr_string ( xdrs,  sp,  maxsize)
XDR *xdrs;
char **sp;
u_int maxsize;
```

Description

The **xdr_string** subroutine is a filter primitive that translates between C language strings and their corresponding external representations. Externally, strings are represented as sequences of ASCII characters, while internally, they are represented with character pointers.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
sp	Specifies the address of the pointer to the string.
maxsize	Specifies the maximum length of the string allowed during encoding or decoding. This value is set in a protocol. For example, if a protocol specifies that a file name cannot be longer than 255 characters, then a string cannot exceed 255 characters.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference
xdr_wrapstring Subroutine
Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_u_char Subroutine

Purpose

Translates between unsigned C language characters and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>

xdr_u_char ( xdrs,  ucp)
XDR *xdrs;
char *ucp;
```

Description

The **xdr_u_char** subroutine is a filter primitive that translates between unsigned C language characters and their external representations.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ucp	Points to an unsigned integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_u_int Subroutine

Purpose

Translates between C language unsigned integers and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdr_u_int (*xdrs*, *up*)
XDR ***xdrs*;
u_int ***up*;

Description

The **xdr_u_int** subroutine is a filter primitive that translates between C language unsigned integers and their external representations.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
up	Specifies the address of the unsigned long integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_u_long Subroutine

Purpose

Translates the unsigned long integers from the C language to their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdr_u_long (*xdrs*, *ulp*)
XDR ***xdrs*;
u_long ***ulp*;

Description

The **xdr_u_long** subroutine is a filter primitive that translates the unsigned long integers from the C language to their external representations.

Note: The `xdr_u_long` subroutine encodes or decodes a 32-bit value, irrespective of whether the application is compiled in 32-bit mode or in 64-bit mode. If a 64-bit value is passed to the `xdr_u_long` subroutine, the resulting high-order 32-bit values are not determined.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
ulp	Specifies the address of the unsigned long integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_u_short Subroutine

Purpose

Translates between C language unsigned short integers and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>

xdr_u_short ( xdrs,  usp)
XDR *xdrs;
u_short *usp;
```

Description

The **xdr_u_short** subroutine is a filter primitive that translates between C language unsigned short integers and their external representations.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
usp	Specifies the address of the unsigned short integer.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_union Subroutine

Purpose

Translates between discriminated unions and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>

xdr_union (xdrs, dscmp, unp, armchoices, defaultarm)
XDR * xdrs;
enum_t * dscmp;
char * unp;
struct xdr_discrim * armchoices;
xdrproc_t (* defaultarm);
```

Description

The **xdr_union** subroutine is a filter primitive that translates between discriminated C unions and their corresponding external representations. It first translates the discriminant of the union located at the address pointed to by the *dscmp* parameter. This discriminant is always an **enum_t** value. Next, this subroutine translates the union located at the address pointed to by the *unp* parameter.

The *armchoices* parameter is a pointer to an array of **xdr_discrim** structures. Each structure contains an ordered pair of parameters [*value, proc*]. If the union's discriminant is equal to the associated value, then the specified process is called to translate the union. The end of the **xdr_discrim** structure array is denoted by a routine having a null value. If the discriminant is not found in the choices array, then the *defaultarm* structure is called (if it is not null).

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
dscmp	Specifies the address of the union's discriminant. The discriminant is an enumeration (**enum_t**) value.
unp	Specifies the address of the union.
armchoices	Points to an array of **xdr_discrim** structures.
defaultarm	A structure provided in case no discriminants are found. This parameter can have a null value.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

xdr_vector Subroutine

Purpose

Translates between fixed-length arrays and their corresponding external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdr_vector (xdrs, arrp, size, elsize, elproc)
XDR * xdrs;
char * arrp;
u_int  size,  elsize;
xdrproc_t  elproc;
```

Description

The **xdr_vector** subroutine is a filter primitive that translates between fixed-length arrays and their corresponding external representations.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
arrp	Specifies the pointer to the array.
size	Specifies the element count of the array.
elsize	Specifies the size of each of the array elements.
elproc	Translates between the C form of the array elements and their external representation. This is an XDR filter.

Return Values

Upon successful completion, this routine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Library Filter Primitives

xdr_void Subroutine

Purpose

Supplies an XDR subroutine to the RPC system without transmitting data.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdr_void ()
```

Description

The **xdr_void** subroutine has no function parameters. It is passed to other Remote Procedure Call (RPC) subroutines that require a function parameter, but does not transmit data.

Return Values

This subroutine always returns a value of 1.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdr_wrapstring Subroutine

Purpose

Calls the **xdr_string** subroutine.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdr_wrapstring ( xdrs,  sp)
XDR *xdrs;
char **sp;
```

Description

The **xdr_wrapstring** subroutine is a primitive that calls the **xdr_string** subroutine (*xdrs, sp, MAXUN.UNSIGNED)*, where the *MAXUN.UNSIGNED* value is the maximum value of an unsigned integer. The **xdr_wrapstring** subroutine is useful because the Remote Procedure Call (RPC) package passes a maximum of two eXternal Data Representation (XDR) subroutines as parameters, and the **xdr_string** subroutine requires three.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.
sp	Specifies the address of the pointer to the string.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference
xdr_string Subroutine
Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming

xdr_authunix_parms Subroutine

Purpose

Describes UNIX-style credentials.

Library

C Library (**libc.a**)

Syntax

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

xdr_authunix_parms ( xdrs,  app)
XDR *xdrs;
struct authunix_parms *app;
```

Description

The **xdr_authunix_parms** subroutine describes UNIX-style credentials. This subroutine generates credentials without using the Remote Procedure Call (RPC) authentication program.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
app	Points to the structure that contains the UNIX-style authentication credentials.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of RPC Programming References
eXternal Data Representation (XDR) Overview for Programming
Remote Procedure Call (RPC) Overview for Programming

xdr_double Subroutine

Purpose

Translates between C language double-precision numbers and their external representations.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdr_double (*xdrs*, *dp*)
XDR ***xdrs*;
double ***dp*;

Description

The **xdr_double** subroutine is a filter primitive that translates between C language double-precision numbers and their external representations.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
dp	Specifies the address of the double-precision number.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Library Filter Primitives

xdrmem_create Subroutine

Purpose

Initializes in local memory the XDR stream pointed to by the *xdrs* parameter.

Library

C Library (**libc.a**)

Syntax

#include <rpc/xdr.h>
void
xdrmem_create (*xdrs*, *addr*, *size*, *op*)
XDR ***xdrs*;
char ***addr*;
u_int *size*;
enum xdr_op *op*;

Description

The **xdrmem_create** subroutine initializes in local memory the eXternal Data Representation (XDR) stream pointed to by the *xdrs* parameter. The XDR stream data is written to or read from a chunk of memory at the location specified by the *addr* parameter.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.
addr	Points to the memory where the XDR stream data is written to or read from.
size	Specifies the length of the memory in bytes.
op	Specifies the XDR direction. The possible choices are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**.

Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

xdrrec_create Subroutine

Purpose

Provides an XDR stream that can contain long sequences of records.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>

void
xdrrec_create (xdrs, sendsize, recvsize, handle, readit, writeit)
XDR * xdrs;
u_int  sendsize;
u_int  recvsize;
char * handle;
int (* readit) (), (* writeit) ();
```

Description

The **xdrrec_create** subroutine provides an eXternal Data Representation (XDR) stream that can contain long sequences of records and handle them in both the encoding and decoding directions. The record contents contain data in XDR form. The routine initializes the XDR stream object pointed to by the *xdrs* parameter.

Note: This XDR stream implements an intermediate record stream. As a result, additional bytes are in the stream to provide record boundary information.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.
sendsize	Sets the size of the input buffer to which data is written. If 0 is specified, the buffers are set to the system defaults.

Item	Description
recvsize	Sets the size of the output buffer from which data is read. If 0 is specified, the buffers are set to the system defaults.
handle	Points to the input/output buffer's handle, which is opaque.
readit	Points to the subroutine to call when a buffer needs to be filled. Similar to the **read** system call.
writeit	Points to the subroutine to call when a buffer needs to be flushed. Similar to the **write** system call.

Related reference
xdrrec_endofrecord Subroutine
xdrrec_eof Subroutine
xdrrec_skiprecord Subroutine
Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

xdrrec_endofrecord Subroutine

Purpose

Causes the current outgoing data to be marked as a record.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

```
xdrrec_endofrecord ( xdrs,  sendnow)
XDR *xdrs;
bool_t sendnow;
```

Description

The **xdrrec_endofrecord** subroutine causes the current outgoing data to be marked as a record and can only be invoked on streams created by the **xdrrec_create** subroutine. If the value of the *sendnow* parameter is nonzero, the data in the output buffer is marked as a completed record and the output buffer is optionally written out.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.
sendnow	Specifies whether the record should be flushed to the output **tcp** stream.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference
xdrrec_create Subroutine
Related information
List of XDR Programming References
Understanding XDR Non-Filter Primitives

xdrrec_eof Subroutine

Purpose

Checks the buffer for an input stream that indicates the end of file (EOF).

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdrrec_eof (*xdrs*)
XDR **xdrs*;

Description

The **xdrrec_eof** subroutine checks the buffer for an input stream to see if the stream reached the end of the file. This subroutine can only be invoked on streams created by the **xdrrec_create** subroutine.

Parameters

Item	Description
xdrs	Points to the eXternal Data Representation (XDR) stream handle.

Return Values

After consuming the rest of the current record in the stream, this subroutine returns a value of 1 if the stream has no more input, and a value of 0 otherwise.

Related reference
xdrrec_create Subroutine
Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming

xdrrec_skiprecord Subroutine

Purpose

Causes the position of an input stream to move to the beginning of the next record.

Library

C Library (**libc.a**)

Syntax

```
#include <rpc/xdr.h>
```

xdrrec_skiprecord (*xdrs*)
XDR ****xdrs*;

Description

The **xdrrec_skiprecord** subroutine causes the position of an input stream to move past the current record boundary and onto the beginning of the next record of the stream. This subroutine can only be invoked on streams created by the **xdrrec_create** subroutine. The **xdrrec_skiprecord** subroutine tells the eXternal Data Representation (XDR) implementation that the rest of the current record in the stream's input buffer should be discarded.

Parameters

Item	Description
xdrs	Points to the XDR stream handle.

Return Values

Upon successful completion, this subroutine returns a value of 1. If unsuccessful, it returns a value of 0.

Related reference
xdrrec_create Subroutine
Related information
List of XDR Programming References
eXternal Data Representation (XDR) Overview for Programming
Understanding XDR Non-Filter Primitives

xdrstdio_create Subroutine

Purpose

Initializes the XDR data stream pointed to by the *xdrs* parameter.

Library

C Library (**libc.a**)

Syntax

```
#include <stdio.h>
#include <rpc/xdr.h>
void xdrstdio_create ( xdrs,  file,  op)
XDR *xdrs;
FILE *file;
enum xdr_op op;
```

Description

The **xdrstdio_create** subroutine initializes the eXternal Data Representation (XDR) data stream pointed to by the *xdrs* parameter. The XDR stream data is written to or read from the standard input/output stream pointed to by the *file* parameter.

Note: The destroy routine associated with such an XDR stream calls the **fflush** function on the *file* stream, but never calls the **fclose** function.

Parameters

Item	Description
xdrs	Points to the XDR stream handle to initialize.
file	Points to the standard I/O device that data is written to or read from.
op	Specifies an XDR direction. The possible choices are **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**.

Related information

List of XDR Programming References

eXternal Data Representation (XDR) Overview for Programming

Understanding XDR Non-Filter Primitives

Network Information Services

This topic collection includes subroutines that derive information from the Network Information Services lookup.

yp_all Subroutine

Purpose

Transfers all of the key-value pairs from the Network Information Services (NIS) server to the client as the entire map.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_all ( indomain,  inmap,  incallback)
char *indomain;
char *inmap;
struct ypall_CallBack *incallback {
int (* foreach) ();
char * data;
};
```

```
foreach (instatus, inkey, inkeylen, inval, invallen, indata)
int  instatus;
char * inkey;
int  inkeylen;
char * inval;
int  invallen;
char * indata;
```

Description

The **yp_all** subroutine provides a way to transfer an entire map from the server to the client in a single request. The routine uses Transmission Control Protocol (TCP) rather than User Datagram Protocol (UDP) used by other NIS subroutines. This entire transaction takes place as a single Remote Procedure Call (RPC) request and response. The **yp_all** subroutine is used like any other NIS procedure, identifying a subroutine and map in the normal manner, and supplying a subroutine to process each key-value pair within the map.

The memory pointed to by the *inkey* and *inval* parameters is private to the **yp_all** subroutine. This memory is overwritten with each new key-value pair processed. The **foreach** function uses the contents of the memory but does not own the memory itself. Key and value objects presented to the **foreach** function look exactly as they do in the server's map. Objects not terminated by a new-line or null character in the server's map are not terminated by a new-line or null character in the client's map.

Note: The remote procedure call is returned to the **yp_all** subroutine only after the transaction is completed (successfully or unsuccessfully) or after the **foreach** function rejects any more key-value pairs.

Parameters

Item	Description
data	Specifies state information between the **foreach** function and the mainline code (see also the *indata* parameter).
indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
incallback	Specifies the structure containing the user-defined **foreach** function, which is called for each key-value pair transferred.
instatus	Specifies either a return status value of the form **NIS_TRUE** or an error code. The error codes are defined in the **rpcsvc/yp_prot.h** file.
inkey	Points to the current key of the key-value pair as returned from the server's database.
inkeylen	Returns the length, in bytes, of the *inkey* parameter.
inval	Points to the current value of the key-value pair as returned from the server's database.
invallen	Specifies the size of the value in bytes.
indata	Specifies the contents of the **incallback->data** element passed to the **yp_all** subroutine. The **data** element shares state information between the **foreach** function and the mainline code. The *indata* parameter is optional because no part of the NIS client package inspects its contents.

Return Values

The **foreach** subroutine returns a value of 0 when it is ready to be called again for additional received key-value pairs. It returns a nonzero value to stop the flow of key-value pairs. If the **foreach** function returns a nonzero value, it is not called again, and the **yp_all** subroutine returns a value of 0.

Related information

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_bind Subroutine

Purpose

Used in programs to call the **ypbind** daemon directly for processes that use backup strategies when Network Information Services (NIS) is not available.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

yp_bind (*indomain*)
char ***indomain;**

Description

In order to use NIS, the client process must be bound to an NIS server that serves the appropriate domain. That is, the client must be associated with a specific NIS server that services the client's requests for NIS information. The NIS lookup processes automatically use the **ypbind** daemon to bind the client, but the **yp_bind** subroutine can be used in programs to call the daemon directly for processes that use backup strategies (for example, a local file) when NIS is not available.

Each NIS binding allocates, or uses up, one client process socket descriptor, and each bound domain uses one socket descriptor. Multiple requests to the same domain use the same descriptor.

Note: If a Remote Procedure Call (RPC) failure status returns from the use of the **yp_bind** subroutine, the domain is unbound automatically. When this occurs, the NIS client tries to complete the operation if the **ypbind** daemon is running and either of the following is true:

- The client process cannot bind a server for the proper domain.
- RPCs to the server fail.

Parameters

Item	Description
indomain	Points to the name of the domain for which to attempt the bind.

Return Values

The NIS client returns control to the user with either an error or a success code if any of the following occurs:

- The error is not related to RPC.
- The **ypbind** daemon is not running.
- The **ypserv** daemon returns the answer.

Related information

ypbind command
Network Information Service (NIS) Overview for System Management
Remote Procedure Call (RPC) Overview for Programming

yp_first Subroutine

Purpose

Returns the first key-value pair from the named Network Information Services (NIS) map in the named domain.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_first (indomain, inmap, outkey, outkeylen, outval, outvallen)
char * indomain;
char * inmap;
char ** outkey;
int * outkeylen;
char ** outval;
int * outvallen;
```

Description

The **yp_first** routine returns the first key-value pair from the named NIS map in the named domain.

Parameters

Item	Description
indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
outkey	Specifies the address of the uninitialized string pointer where the first key is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.
outkeylen	Returns the length, in bytes, of the *outkey* parameter.
outval	Specifies the address of the uninitialized string pointer where the value associated with the key is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.
outvallen	Returns the length, in bytes, of the *outval* parameter.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns an error as described in the **rpcsvc/yp_prot.h** file.

Related information

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_get_default_domain Subroutine

Purpose

Gets the default domain of the node.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

yp_get_default_domain (*outdomain*)
**char *****outdomain*;

Description

Network Information Services (NIS) lookup calls require both a map name and a domain name. Client processes can get the default domain of the node by calling the **yp_get_default_domain** routine and using the value returned in the *outdomain* parameter as the input domain *(indomain*) parameter for NIS remote procedure calls.

Parameters

Item	Description
outdomain	Specifies the address of the uninitialized string pointer where the default domain is returned. Memory is allocated by the NIS client using the **malloc** subroutine and should not be freed by the application.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns an error as described in the **rpcsvc/ypclnt.h** file.

Related information
malloc subroutine
Network Information Service (NIS) Overview for System Management
Remote Procedure Call (RPC) Overview for Programming

yp_master Subroutine

Purpose

Returns the machine name of the Network Information Services (NIS) master server for a map.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_master ( indomain,  inmap,  outname)
char *indomain;
char *inmap;
char **outname;
```

Description

The **yp_master** subroutine returns the machine name of the NIS master server for a map.

Parameters

Item	Description
indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
outname	Specifies the address of the uninitialized string pointer where the name of the domain's **yp_master** server is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_match Subroutine

Purpose

Searches for the value associated with a key.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_match (indomain, inmap, inkey, inkeylen, outval, outvallen)
char * indomain;
char * inmap;
char * inkey;
int  inkeylen;
char ** outval;
int * outvallen;
```

Description

The **yp_match** subroutine searches for the value associated with a key. The input character string entered as the key must match a key in the Network Information Services (NIS) map exactly because pattern matching is not available in NIS.

Parameters

Item	Description
indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
inkey	Points to the name of the key used as input to the subroutine.
inkeylen	Specifies the length, in bytes, of the key.
outval	Specifies the address of the uninitialized string pointer where the values associated with the key are returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.
outvallen	Returns the length, in bytes, of the *outval* parameter.

Return Values

Upon successful completion, this subroutine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_next Subroutine

Purpose

Returns each subsequent value it finds in the named Network Information Services (NIS) map until it reaches the end of the list.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

```
yp_next (indomain, inmap, inkey, inkeylen, outkey, outkeylen, outval,
outvallen)
char * indomain;
char * inmap;
char * inkey;
int  inkeylen;
char ** outkey;
int * outkeylen;
char ** outval;
int * outvallen;
```

Description

The **yp_next** subroutine returns each subsequent value it finds in the named NIS map until it reaches the end of the list.

The **yp_next** subroutine must be preceded by an initial **yp_first** subroutine. Use the *outkey* parameter value returned from the initial **yp_first** subroutine as the value of the *inkey* parameter for the **yp_next** subroutine. This will return the second key-value pair associated with the map. To show every entry in the NIS map, the **yp_first** subroutine is called with the **yp_next** subroutine called repeatedly. Each time the **yp_next** subroutine returns a key-value, use it as the *inkey* parameter for the next call.

The concepts of *first* and *next* depend on the structure of the NIS map being processed. The routines do not retrieve the information in a specific order, such as the lexical order from the original, non-NIS database information files or the numerical sorting order of the keys, values, or key-value pairs. If the **yp_first** subroutine is called on a specific map with the **yp_next** subroutine called repeatedly until the process returns a **YPERR_NOMORE** message, every entry in the NIS map is seen once. If the same sequence of operations is performed on the same map at the same server, the entries are seen in the same order.

Note: If a server operates under a heavy load or fails, the domain can become unbound and then bound again while a client is running. If it binds itself to a different server, entries may be seen twice or not at all. The domain rebinds itself to protect the enumeration process from being interrupted before it completes. Avoid this situation by returning all of the keys and values with the **yp_all** subroutine.

Parameters

Item	Description
indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
inkey	Points to the key that is used as input to the subroutine.
inkeylen	Returns the length, in bytes, of the *inkey* parameter.
outkey	Specifies the address of the uninitialized string pointer where the first key is returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.
outkeylen	Returns the length, in bytes, of the *outkey* parameter.
outval	Specifies the address of the uninitialized string pointer where the values associated with the key are returned. Memory is allocated by the NIS client using the **malloc** subroutine, and may be freed by the application.
outvallen	Returns the length, in bytes, of the *outval* parameter.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information

malloc subroutine

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_order Subroutine

Purpose

Returns the order number for an Network Information Services (NIS) map that identifies when the map was built.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

yp_order (*indomain*, *inmap*, *outorder*)
char * *indomain*;
char * *inmap*;
int * *outorder*;

Description

The **yp_order** subroutine returns the order number for a NIS map that identifies when the map was built. The number determines whether the local NIS map is more current than the master NIS database.

Parameters

Item	Description
indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
outorder	Points to the returned order number, which is a 10-digit ASCII integer that represents the operating system time, in seconds, when the map was built.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yp_unbind Subroutine

Purpose

Manages socket descriptors for processes that access multiple domains.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

void yp_unbind (*indomain*)
char ****indomain*;

Description

The **yp_unbind** subroutine is available to manage socket descriptors for processes that access multiple domains. When the **yp_unbind** subroutine is used to free a domain, all per-process and per-node resources that were used to bind the domain are also freed.

Parameters

Item	Description
indomain	Points to the name of the domain used as input to the subroutine.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Related information
ypbind command
Remote Procedure Call (RPC) Overview for Programming
Sockets Overview

yp_update Subroutine

Purpose

Makes changes to an Network Information Services (NIS) map.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

yp_update (*indomain*, *inmap*, *ypop*, *inkey*, *inkeylen*, *indata*, *indatalen*)
char * *indomain*;
char * *inmap*;
unsigned *ypop*;
char * *inkey*;
int *inkeylen*;
char * *indata*;
int *indatalen*;

Description

Note: This routine depends upon the secure Remote Procedure Call (RPC) protocol, and will not work unless the network is running it.

The **yp_update** subroutine is used to make changes to a NIS map. The syntax is the same as that of the **yp_match** subroutine except for the additional *ypop* parameter, which may take on one of the following four values:

Value	Description
ypop _INSERT	Inserts the key-value pair into the map. If the key already exists in the map, the **yp_update** subroutine returns a value of **YPERR_KEY**.
ypop_CHANGE	Changes the data associated with the key to the new value. If the key is not found in the map, the **yp_update** subroutine returns a value of **YPERR_KEY**.
ypop_STORE	Stores an item in the map regardless of whether the item already exists. No error is returned in either case.
ypop_DELETE	Deletes an entry from the map.

Parameters

Item	Description
indomain	Points to the name of the domain used as input to the subroutine.
inmap	Points to the name of the map used as input to the subroutine.
ypop	Specifies the update operation to be used as input to the subroutine.
inkey	Points to the input key to be used as input to the subroutine.
inkeylen	Specifies the length, in bytes, of the *inkey* parameter.
indata	Points to the data used as input to the subroutine.
indatalen	Specifies the length, in bytes, of the data used as input to the subroutine.

Return Values

Upon successful completion, this routine returns a value of 0. If unsuccessful, it returns one of the error codes described in the **rpcsvc/yp_prot.h** file.

Files

Item	Description
/var/yp/updaters	A makefile for updating NIS maps.

Related information

Network Information Service (NIS) Overview for System Management

Remote Procedure Call (RPC) Overview for Programming

yperr_string Subroutine

Purpose

Returns a pointer to an error message string.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

char *yperr_string (*incode*)
int *incode*;

Description

The **yperr_string** routine returns a pointer to an error message string. The error message string is null-terminated but contains no period or new-line escape characters.

Parameters

Item	Description
incode	Contains Network Information Services (NIS) error codes as described in the **rpcsvc/yp_prot.h** file.

Return Values

This subroutine returns a pointer to an error message string corresponding to the *incode* parameter.

Related reference
ypprot_err Subroutine
Related information
Network Information Service (NIS) Overview for System Management

ypprot_err Subroutine

Purpose

Takes an Network Information Services NIS protocol error code as input and returns an error code to be used as input to a **yperr_string** subroutine.

Library

C Library (**libc.a**)

Syntax

```
#include <rpcsvc/ypclnt.h>
#include <rpcsvc/yp_prot.h>
```

ypprot_err (*incode*)
u_int *incode*;

Description

The **ypprot_err** subroutine takes a NIS protocol error code as input and returns an error code to be used as input to a **yperr_string** subroutine.

Parameters

Item	Description
incode	Specifies the NIS protocol error code used as input to the subroutine.

Return Values

This subroutine returns a corresponding error code to be passed to the **yperr_string** subroutine.

Related reference
yperr_string Subroutine
Related information
Network Information Service (NIS) Overview for System Management
Remote Procedure Call (RPC) Overview for Programming

Network Information Services+

This topic collection includes subroutines that are used for network information services+ (NIS+).

nis_add_entry (NIS+ API)

Purpose

Used to add the NIS+ object to the NIS+ table_name.

Syntax

cc [***flag*** . . .] ***file***. . . **-lnsl** [***library. . .***]

```
#include <rpcsvc/nis.h>
```

nis_result * nis_add_entry(nis_name *table_name*, nis_object object, u_long* *flags*);

Description

One of a group of NIS+ APIs that is used to search and modify NIS+ tables, **nis_add_entry( )** is used to add the NIS+ object to the NIS+ ***table_name.***.

Entries within a table are named by NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket **[ ]** characters. Indexed names have the following form:

```
[ colname=value,...],tablename
```

nis_add_entry( ) will add the NIS+ object to the NIS+ ***table_name***. The ***flags*** parameter is used to specify the failure semantics for the add operation:

0
The default (***flags*** = 0) is to fail if the entry being added already exists in the table.

ADD_OVERWRITE
Specifies that the existing object is to be overwritten if it exists (a modify operation), or added if it does not exist. With the **ADD_OVERWRITE** flag, this function will fail with the error **NIS_PERMISSION** if the existing object does not allow modify privileges to the client.

RETURN_RESULT
Specifies that the server will return a copy of the resulting object if the operation was successful. To succeed, **nis_add_entry( )**must inherit the **PAF_TRUSTED_PATH** attribute.

Return Values

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
          nis_error status;
```

```
        struct {
                  u_int        objects_len;
                  nis_object   * objects_val;
        } objects;
        netobj     cookie;
        u_long     zticks;
        u_long     dticks;
        u_long     aticks;
        u_long     cticks;
};
```

The ***status*** member contains the error status of the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno( )**.

The **objects** structure contains two members: ***objects_val*** is an array of ***nis_object*** structures; ***objects_len*** is the number of cells in the array. These objects will be freed by a call to **nis_freeresult( )**. If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object( )** and freed with the function **nis_destroy_object( )**.

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

zticks
The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks
The time spent in the database backend, this time is measured from the time a database call starts until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks
The time spent in any *accelerators* or caches. This includes the time required to locate the server needed to resolve the request.

cticks
The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating an NIS+ request.

Subtracting the value in ***dticks*** from the value in ***zticks*** will yield the time spent in the service code itself. Subtracting the sum of the values in ***zticks*** and ***aticks*** from the value in ***cticks*** will yield the time spent in the client library itself.

Note: All of the tick times are measured in microseconds.

Errors

The client library can return a variety of error returns and diagnostics. Following are some of the more pertinent ones:

NIS_BADATTRIBUTE
The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value.

NIS_BADNAME
The name passed to the function is not a legal NIS+ name.

NIS_BADREQUEST
A problem was detected in the request structure passed to the client library.

NIS_CACHEEXPIRED
The entry returned came from an object cache that has ***expired***. This means that the time to live value has gone to zero and the entry may have changed. If the flag **NO_CACHE** was passed to the lookup function, the lookup function will retry the operation to get an unexpired copy of the object.

NIS_CBERROR
An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded.

NIS_CBRESULTS
Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result.

NIS_FOREIGNNS
The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ
The object pointed to by ***object*** is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL
The attempted modification failed.

NIS_NAMEEXISTS
An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOCALLBACK
The server was unable to contact the callback service on your machine. This results in no data being returned.

NIS_NOMEMORY
Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME
This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table does not know about the directory in which the table resides.

NIS_NOSUCHTABLE
The named table does not exist.

NIS_NOT_ME
A request was made to a server that does not serve the given name. Normally this will not occur; however, if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND
No entries in the table matched the search criteria. If the search criteria was null (return all entries), then this result means that the table is empty and may safely be removed by calling the **nis_remove( )**. If the **FOLLOW_PATH** flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria.

NIS_NOTMASTER
A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the **/var/nis/NIS_SHARED_DIRCACHE** file will need to have their cache managers restarted (use **nis_cachemgr -i** to flush this cache).

NIS_NOTSAMEOBJ
An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOTSEARCHABLE
The table name resolved to a NIS+ object that was not searchable.

NIS_PARTIAL
This result is similar to **NIS_NOTFOUND**, except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy.

NIS_RPCERROR
This unrecoverable error indicates the RPC subsystem failed in some way. Generally there will be a **syslog**(3) message indicating why the RPC request failed.

NIS_S_NOTFOUND
The named entry does not exist in the table; however, not all tables in the path could be searched, so the entry may exist in one of those tables.

NIS_S_SUCCESS
Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible.

NIS_SUCCESS
The request was successful.

NIS_SYSTEMERROR
Some form of generic system error occurred while attempting the request. Check the **syslog**(3) record for error messages from the server.

NIS_TOOMANYATTRS
The search criteria passed to the server had more attributes than the table had searchable columns.

NIS_TRYAGAIN
The server connected to was too busy to handle your request. **add_entry( )**, **remove_entry( )**, and **modify_entry( )** return this error when the master server is currently updating its internal state. It can be returned to **nis_list( )** when the function specifies a callback and the server does not have the resources to handle callbacks.

NIS_TYPEMISMATCH
An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table.

Summary of Trusted

To succeed, **nis_add_entry( )** must inherit the **PAF_TRUSTED_PATH** attribute.

nis_first_entry (NIS+ API)

Purpose

Used to fetch entries from a table one at a time.

Syntax

cc [***flag*** . . .] ***file***. . . **-lnsl** [***library. . .***]

```
#include <rpcsvc/nis.h>
```

nis_result * nis_first_entry(nis_name *table_name*)

Description

One of a group of NIS+ APIs that is used to search and modify NIS+ tables, **nis_first_entry( )** is used to fetch entries from a table one at a time.

Entries within a table are named by .NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket **[ ]** characters. Indexed names have the following form:

```
[ colname=value,...],tablename
```

nis_first_entry( ) fetches entries from a table one at a time. This mode of operation is extremely inefficient and callbacks should be used instead whenever possible. The table containing the entries of interest is identified by ***name***. If a search criteria is present in ***name*** it is ignored. The value of ***cookie*** within the **nis_result** structure must be copied by the caller into local storage and passed as an argument to **nis_next_entry( )**.

Return Values

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
           nis_error status;
           struct {
                      u_int        objects_len;
                      nis_object   * objects_val;
           } objects;
           netobj     cookie;
           u_long     zticks;
           u_long     dticks;
           u_long     aticks;
           u_long     cticks;
     };
```

The ***status*** member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno( )**.

The **objects** structure contains two members: ***objects_val*** is an array of ***nis_object*** structures; ***objects_len*** is the number of cells in the array. These objects will be freed by a call to **nis_freeresult( )**. If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object( )** and freed with the function **nis_destroy_object( )**.

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

zticks
: The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks
: The time spent in the database backend, this time is measured from the time a database call starts, until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks
: The time spent in any *accelerators* or caches. This includes the time required to locate the server needed to resolve the request.

cticks
: The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating an NIS+ request.

Subtracting the value in ***dticks*** from the value in ***zticks*** will yield the time spent in the service code itself. Subtracting the sum of the values in ***zticks*** and ***aticks*** from the value in ***cticks*** will yield the time spent in the client library itself.

Note: All of the tick times are measured in microseconds.

Errors

The client library can return a variety of error returns and diagnostics. Following are some of the more pertinent ones:

NIS_BADATTRIBUTE
The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value.

NIS_BADNAME
The name passed to the function is not a legal NIS+ name.

NIS_BADREQUEST
A problem was detected in the request structure passed to the client library.

NIS_CACHEEXPIRED
The entry returned came from an object cache that has ***expired***. This means that the time to live value has gone to zero and the entry may have changed. If the flag **NO_CACHE** was passed to the lookup function, the lookup function will retry the operation to get an unexpired copy of the object.

NIS_CBERROR
An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded.

NIS_CBRESULTS
Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result.

NIS_FOREIGNNS
The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with aNIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ
The object pointed to by ***object*** is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL
The attempted modification failed.

NIS_NAMEEXISTS
An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOCALLBACK
The server was unable to contact the callback service on your machine. This results in no data being returned.

NIS_NOMEMORY
Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME
This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table does not know about the directory in which the table resides.

NIS_NOSUCHTABLE
The named table does not exist.

NIS_NOT_ME
A request was made to a server that does not serve the given name. Normally this will not occur; however, if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND
No entries in the table matched the search criteria. If the search criteria was null (return all entries), then this result means that the table is empty and may safely be removed by calling the **nis_remove( )**. If the **FOLLOW_PATH** flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria.

NIS_NOTMASTER
A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the **/var/nis/NIS_SHARED_DIRCACHE** file will need to have their cache managers restarted (use **nis_cachemgr -i** to flush this cache).

NIS_NOTSAMEOBJ
An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOTSEARCHABLE
The table name resolved to a NIS+ object that was not searchable.

NIS_PARTIAL
This result is similar to **NIS_NOTFOUND**, except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy.

NIS_RPCERROR
This fatal error indicates the RPC subsystem failed in some way. Generally there will be a **syslog**(3) message indicating why the RPC request failed.

NIS_S_NOTFOUND
The named entry does not exist in the table; however, not all tables in the path could be searched, so the entry may exist in one of those tables.

NIS_S_SUCCESS
Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible.

NIS_SUCCESS
The request was successful.

NIS_SYSTEMERROR
Some form of generic system error occurred while attempting the request. Check the **syslog**(3) record for error messages from the server.

NIS_TOOMANYATTRS
The search criteria passed to the server had more attributes than the table had searchable columns.

NIS_TRYAGAIN
The server connected to was too busy to handle your request. **add_entry( )**, **remove_entry( )**, and **modify_entry( )** return this error when the master server is currently updating its internal state. It can be returned to **nis_list( )** when the function specifies a callback and the server does not have the resources to handle callbacks.

NIS_TYPEMISMATCH
An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table.

nis_list (NIS+ API)

Purpose

Used to search a table in the NIS+ namespace.

Syntax

cc [***flag*** . . .] ***file***. . . **-lnsl** [***library. . .***]

```
#include <rpcsvc/nis.h>
```

nis_result * nis_list(*name*, *flags*, *callback userdata*);

```
nis_name  name;
```

```
u_long  flags;
```

```
int  (*callback)( );
```

```
void  userdata;
```

Description

One of a group of NIS+ APIs that is used to search and modify NIS+ tables, **nis_list( )** is used to search a table in the NIS+ ***namespace.***.

Entries within a table are named by NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket **[ ]** characters. Indexed names have the following form:

```
[ colname=value,...],tablename
```

The list function, **nis_list( )**, takes an indexed name as the value for the ***name*** parameter. Here, the tablename should be a fully qualified NIS+ name unless the **EXPAND_NAME** flag is set. The second parameter, ***flags***, defines how the function will respond to various conditions. The value for this parameter is created by logically **OR** ing together one or more flags from the following list:

FOLLOW_LINKS
If the table specified in ***name*** resolves to be a **LINK** type object, this flag specifies that the client library follow that link and do the search at that object. If this flag is not set and the name resolves to a link, the error **NIS_NOTSEARCHABLE** will be returned.

FOLLOW_PATH
This flag specifies that if the entry is not found within this table, the list operation should follow the path specified in the table object. When used in conjunction with the **ALL_RESULTS** flag, it specifies that the path should be followed regardless of the result of the search. When used in conjunction with the **FOLLOW_LINKS** flag, named tables in the path that resolve to links will be followed until the table they point to is located. If a table in the path is not reachable because no server that serves it is available, the result of the operation will be either a "soft" success or a "soft" failure to indicate that not all tables in the path could be searched. If a name in the path names is either an invalid or non-existent object, then it is silently ignored.

HARD_LOOKUP
This flag specifies that the operation should continue trying to contact a server of the named table until a definitive result is returned (such as **NIS_NOTFOUND**).

> **Warning:** Use the flag **HARD_LOOKUP** carefully since it can cause the application to block indefinitely during a network partition.

ALL_RESULTS
This flag can only be used in conjunction with **FOLLOW_PATH** and a callback function. When specified, it forces all of the tables in the path to be searched. If ***name*** does not specify a search criteria (imply that all entries are to be returned), then this flag will cause all of the entries in all of the tables in the path to be returned.

NO_CACHE
This flag specifies that the client library should bypass any client object caches and get its information directly from either the master server or a replica server for the named table.

MASTER_ONLY
This flag is even stronger than **NO_CACHE** as it specifies that the client library should ***only*** get its information from the master server for a particular table. This guarantees that the information will be up-to-date. However, there may be severe performance penalties associated with contacting the master server directly on large networks. When used in conjunction with the **HARD_LOOKUP** flag, this will block the list operation until the master server is up and available.

EXPAND_NAME
When specified, the client library will attempt to expand a partially qualified name by calling **nis_getnames( )**, which uses the environment variable **NIS_PATH**.

RETURN_RESULT
This flag is used to specify that a copy of the returning object be returned in the **nis_result** structure if the operation was successful.

The third parameter to **nis_list( )**, ***callback***, is an optional pointer to a function that will process the **ENTRY** type objects that are returned from the search. If this pointer is **NULL**, then all entries that match the search criteria are returned in the ***nis_result*** structure; otherwise, this function will be called once for each entry returned. When called, this function should return **0** when additional objects are desired, and **1** when it no longer wishes to see any more objects.

The fourth parameter, ***userdata***, is simply passed to callback function along with the returned entry object. The client can use this pointer to pass state information or other relevant data that the callback function might need to process the entries.

Return Values

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
           nis_error status;
           struct {
                      u_int        objects_len;
                      nis_object   * objects_val;
           } objects;
           netobj     cookie;
           u_long     zticks;
           u_long     dticks;
           u_long     aticks;
           u_long     cticks;
     };
```

The ***status*** member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno( )**.

The **objects** structure contains two members: ***objects_val*** is an array of ***nis_object*** structures; ***objects_len*** is the number of cells in the array. These objects will be freed by a call to **nis_freeresult( )**. If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object( )** and freed with the function **nis_destroy_object( )**.

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

zticks
The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks
The time spent in the database backend, this time is measured from the time a database call starts, until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks
The time spent in any *accelerators* or caches. This includes the time required to locate the server needed to resolve the request.

cticks
The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating anNIS+ request.

Subtracting the value in ***dticks*** from the value in ***zticks*** will yield the time spent in the service code itself. Subtracting the sum of the values in ***zticks*** and ***aticks*** from the value in ***cticks*** will yield the time spent in the client library itself.

Note: All of the tick times are measured in microseconds.

Errors

The client library can return a variety of error returns and diagnostics. Following are some of the more pertinent ones:

NIS_BADATTRIBUTE
The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value.

NIS_BADNAME
The name passed to the function is not a legal NIS+ name.

NIS_BADREQUEST
A problem was detected in the request structure passed to the client library.

NIS_CACHEEXPIRED
The entry returned came from an object cache that has ***expired***. This means that the time to live value has gone to zero and the entry may have changed. If the flag **NO_CACHE** was passed to the lookup function, the lookup function will retry the operation to get an unexpired copy of the object.

NIS_CBERROR
An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded.

NIS_CBRESULTS
Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result.

NIS_FOREIGNNS
The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ
The object pointed to by ***object*** is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL
The attempted modification failed.

NIS_NAMEEXISTS
An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOCALLBACK
The server was unable to contact the callback service on your machine. This results in no data being returned.

NIS_NOMEMORY
Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME
This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table does not know about the directory in which the table resides.

NIS_NOSUCHTABLE
The named table does not exist.

NIS_NOT_ME
A request was made to a server that does not serve the given name. Normally this will not occur; however, if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND
No entries in the table matched the search criteria. If the search criteria was null (return all entries), then this result means that the table is empty and may safely be removed by calling the **nis_remove( )**. If the **FOLLOW_PATH** flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria.

NIS_NOTMASTER
A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the **/var/nis/NIS_SHARED_DIRCACHE** file will need to have their cache managers restarted (use **nis_cachemgr -i** to flush this cache).

NIS_NOTSAMEOBJ
An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOTSEARCHABLE
The table name resolved to a NIS+ object that was not searchable.

NIS_PARTIAL
This result is similar to **NIS_NOTFOUND** except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy.

NIS_RPCERROR
This fatal error indicates the RPC subsystem failed in some way. Generally there will be a **syslog**(3) message indicating why the RPC request failed.

NIS_S_NOTFOUND
The named entry does not exist in the table; however, not all tables in the path could be searched, so the entry may exist in one of those tables.

NIS_S_SUCCESS
Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible.

NIS_SUCCESS
The request was successful.

NIS_SYSTEMERROR
Some form of generic system error occurred while attempting the request. Check the **syslog**(3) record for error messages from the server.

NIS_TOOMANYATTRS
The search criteria passed to the server had more attributes than the table had searchable columns.

NIS_TRYAGAIN
The server connected to was too busy to handle your request. **add_entry( )**, **remove_entry( )**, and **modify_entry( )** return this error when the master server is currently updating its internal state. It can be returned to **nis_list( )** when the function specifies a callback and the server does not have the resources to handle callbacks.

NIS_TYPEMISMATCH
An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table.

Environment

NIS_PATH
When set, this variable is the search path used by **nis_list( )** if the flag **EXPAND_NAME** is set.

Notes:

- The path used when the flag **FOLLOW_PATH** is specified is the one present in the ***first*** table searched. The path values in tables that are subsequently searched are ignored.
- It is legal to call functions that would access the nameservice from within a list callback. However, calling a function that would itself use a callback, or calling **nis_list( )** with a callback from within a list callback function, is not currently supported.

nis_local_directory (NIS+ API)

Purpose

Returns the name of the NIS+ domain for this machine.

Syntax

cc [***flag*** . . .] ***file***. . . **-lnsl** [***library. . .***]

```
#include <rpcsvc/nis.h>
```

nis_name nis_local_directory(void)

Description

One of a group of NIS+ APIs that return several default NIS+ names associated with the current process, **nis_local_directory( )** returns the name of the NIS+ domain for this machine. This is currently the same as the Secure RPC domain returned by the **sysinfo**(2) system call.

Note: The result returned by this routine is a pointer to a data structure with the NIS+ library, and should be considered a "read-only" result and should not be modified.

Environment

nis_group
This variable contains the name of the local NIS+ group. If the name is not fully qualified, the value returned by **nis_local_directory( )** will be concatenated to it.

nis_lookup (NIS+ API)

Purpose

Used to resolve an NIS+ name and return a copy of that object from an NIS+ server.

Syntax

cc [***flag*** . . .] ***file***. . . **-lnsl** [***library. . .***]

```
#include <rpcsvc/nis.h>
```

```
nis_result * nis_lookup(nis_name name, u_long flags);
```

```
void nis_freeresult(nis_result * result);
```

Description

One of a group of NIS+ APIs that is used to locate and manipulate all NIS+ objects except the NIS+ entry objects, **nis_lookup( )** resolves an NIS+ name and returns a copy of that object from an NIS+ server.

This function should be used only with names that refer to an NIS+Directory, NIS+Table, NIS+Group, or NIS+Private object. If a name refers to an NIS+ entry object, the functions listed in **nis_subr(3N)** should be used.

nis_lookup returns a pointer to a **nis_result structure** that ***must*** be freed by calling **nis_freeresult( )** when you have finished using it. If one or more of the objects returned in the structure need to be retained, they can be copied with **nis_clone_object(3N)**.

nis_lookup( ) takes two parameters, the name of the object to be resolved in ***name***, and a flags parameter, ***flags***. The object name is expected to correspond to the syntax of a non-indexed NIS+ name. The **nis_lookup( )** function is the ***only*** function from this group that can use a non-fully qualified name. If the parameter ***name*** is not a fully qualified name, then the flag **EXPAND_NAME *must*** be specified in the call. If this flag is not specified, the function will fail with the error **NIS+BADNAME**.

The ***flags*** parameter is constructed by logically **OR** ing zero or more flags from the following list:

EXPAND_NAME
When specified, the client library will attempt to expand a partially qualified name by calling the function **nis_getnames( )**, which uses the environment variable **nis_path**.

FOLLOW_LINKS
When specified, the client library will "follow" links by issuing another NIS+ lookup call for the object named by the link. If the linked object is itself a link, then this process will iterate until either an object is found that is not a **link** type object, or the library has followed 16 links.

HARD_LOOKUP
When specified, the client library will retry the lookup until it is answered by a server. Using this flag will cause the library to block until at least one NIS+ server is available. If the network connectivity is impaired, this can be a relatively long time.

MASTER_ONLY
When specified, the client library will bypass any object caches and any domain replicas and fetch the object from the NIS+ master server for the object's domain. This insures that the object returned is up-to-date at the cost of a possible performance degradation and failure if the master server is unavailable or physically distant.

NO_CACHE
When specified, the client library will bypass any object caches and will get the object from either the master NIS+ server or one of its replicas.

The status value may be translated to ascii text using the function **nis_sperrno( )**.

On return, the ***objects*** array in the result will contain one and possibly several objects that were resolved by the request. If the **FOLLOW_LINKS** flag was present, on success the function could return several entry objects if the link in question pointed within a table. If an error occurred when following a link, the objects array will contain a copy of the link object itself.

Return Values

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
           nis_error status;
           struct {
                      u_int objects_len;
                      nis_object * objects_val;
           } objects;
           netobj     cookie;
           u_long     zticks;
           u_long     dticks;
           u_long     aticks;
           u_long     cticks;
     };
```

The ***status*** member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno( )**.

The **objects** structure contains two members: ***objects_val*** is an array of ***nis_object*** structures; ***objects_len***is the number of cells in the array. These objects will be freed by a call to **nis_freeresult( )**. If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object( )** and freed with the function **nis_destroy_object( )**.

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

zticks
The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks
The time spent in the database backend, this time is measured from the time a database call starts, until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks
The time spent in any *accelerators* or caches. This includes the time required to locate the server needed to resolve the request.

cticks
The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating an NIS+ request.

Subtracting the value in ***dticks*** from the value in ***zticks*** will yield the time spent in the service code itself. Subtracting the sum of the values in ***zticks*** and ***aticks*** from the value in ***cticks*** will yield the time spent in the client library itself.

Note: All of the tick times are measured in microseconds.

Errors

The client library can return a variety of error returns and diagnostics. Following are some of the more pertinent ones:

NIS_BADNAME
The name passed to the function is not a legal **NIS+** name.

NIS_CACHEEXPIRED
The object returned came from an object cache that has ***expired***. This means that the time to live value has gone to zero and the entry may have changed. If the flag **NO_CACHE** was passed to the lookup function, the lookup function will retry the operation to get an unexpired copy of the object.

NIS_FOREIGNNS
The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ
The object pointed to by ***obj*** is not a valid **NIS+** object.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL
The attempted modification failed.

NIS_NAMEEXISTS
An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
A server for the directory of the named object could not be reached. This can occur when there is a network partition or all the server s have crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOMEMORY
Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME
This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table does not know about the directory in which the table resides.

NIS_NOSUCHTABLE
The named table does not exist.

NIS_NOT_ME
A request was made to a server that does not serve the given name. Normally this will not occur; however, if you are not using the built-in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND
The named object does not exist in the namespace.

NIS_NOTMASTER
An attempt was made to update the database on a replica server.

NIS_NOTSAMEOBJ
An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_RPCERROR
This fatal error indicates the RPC subsystem failed in some way. Generally there will be a **syslog**(3) message indicating why the RPC request failed.

NIS_S_SUCCESS
The request was successful; however, the object returned came from an object cache and not directly from the server. If you want to see objects from object caches, you must specify the flag **NO_CACHE** when you call the lookup function.

NIS_SUCCESS
The request was successful.

NIS_SYSTEMERROR
A generic system error occurred while attempting the request. Most commonly the server has crashed or the database has become corrupted. Check the **syslog** record for error messages from the server.

NIS_TRYAGAIN
The server connected to was too busy to handle your request. For the ***add***, ***remove***, and ***modify*** operations this is returned when either the master server for a directory is unavailable or it is in the process of checkpointing its database. It can also be returned when the server is updating its internal state or, in the case of **nis_list( )**, if the client specifies a callback and the server does not have the resources to handle callbacks.

NIS_UNKNOWNOBJ
The object returned is of an unknown type.

Environment

NIS_PATH
If the flag **EXPAND_NAME** is set, this variable is the search path used by **nis_lookup( )**.

nis_modify_entry (NIS+ API)

Purpose

Used to modify an NIS+ object identified by ***name***.

Syntax

cc [***flag*** . . .] ***file***. . . **-lnsl** [***library. . .***]

```
#include <rpcsvc/nis.h>
```

nis_remove_entry * nis_remove_entry(nis_name *name*, nis_object * *object*, u_long *flags*);

Description

One of a group of NIS+ APIs that is used to search and modify NIS+ tables; **nis_modify_entry( )** is used to remove the identified entry from the table or a set of entries identified by ***table_name***.

Entries within a table are named by NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket **[ ]** characters. Indexed names have the following form:

```
[ colname=value,...],tablename
```

nis_modify_entry( ) modifies an object identified by ***name***. The parameter ***object*** should point to an entry with the **EN_MODIFIED** flag set in each column that contains new information.

The owner, group, and access rights of an entry are modified by placing the modified information into the respective fields of the parameter, ***object***: **zo_owner**, **zo_group**, and **zo_access**.

These columns will replace their counterparts in the entry that is stored in the table. The entry passed must have the same number of columns, same type, and valid data in the modified columns for this operation to succeed.

If the flags parameter contains the flag **MOD_SAMEOBJ**, the object pointed to by ***object*** is assumed to be a cached copy of the original object. If the OID of the object passed is different than the OID of the object the server fetches, then the operation fails with the **NIS_NOTSAMEOBJ** error. This can be used to implement a simple read-modify-write protocol that will fail if the object is modified before the client can write the object back.

If the flag **RETURN_RESULT** has been specified, the server will return a copy of the resulting object if the operation was successful.

To succeed, **nis_modify_entry( )** must inherit the **PAF_TRUSTED_PATH** attribute.

Return Values

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
          nis_error status;
          struct {
                    u_int        objects_len;
                    nis_object   * objects_val;
          } objects;
          netobj     cookie;
          u_long     zticks;
          u_long     dticks;
          u_long     aticks;
          u_long     cticks;
     };
```

The ***status*** member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno( )**.

The **objects** structure contains two members: ***objects_val*** is an array of ***nis_object*** structures; ***objects_len*** is the number of cells in the array. These objects will be freed by a call to **nis_freeresult( )**. If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object( )** and freed with the function **nis_destroy_object( )**.

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

zticks
: The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks
: The time spent in the database backend, this time is measured from the time a database call starts, until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks
: The time spent in any *accelerators* or caches. This includes the time required to locate the server needed to resolve the request.

cticks
: The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating an NIS+ request.

Subtracting the value in ***dticks*** from the value in ***zticks*** will yield the time spent in the service code itself. Subtracting the sum of the values in ***zticks*** and ***aticks*** from the value in ***cticks*** will yield the time spent in the client library itself.

Note: All of the tick times are measured in microseconds.

Errors

The client library can return a variety of error returns and diagnostics. Following are some of the more pertinent ones:

NIS_BADATTRIBUTE
: The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value.

NIS_BADNAME
The name passed to the function is not a legal NIS+ name.

NIS_BADREQUEST
A problem was detected in the request structure passed to the client library.

NIS_CACHEEXPIRED
The entry returned came from an object cache that has ***expired***. This means that the time to live value has gone to zero and the entry may have changed. If the flag **NO_CACHE** was passed to the lookup function, the lookup function will retry the operation to get an unexpired copy of the object.

NIS_CBERROR
An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded.

NIS_CBRESULTS
Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result.

NIS_FOREIGNNS
The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ
The object pointed to by ***object*** is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL
The attempted modification failed.

NIS_NAMEEXISTS
An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOCALLBACK
The server was unable to contact the callback service on your machine. This results in no data being returned.

NIS_NOMEMORY
Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME
This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table does not know about the directory in which the table resides.

NIS_NOSUCHTABLE
The named table does not exist.

NIS_NOT_ME
A request was made to a server that does not serve the given name. Normally, this will not occur; however, if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND
No entries in the table matched the search criteria. If the search criteria was null (return all entries), then this result means that the table is empty and may safely be removed by calling the **nis_remove( )**. If the **FOLLOW_PATH** flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria.

NIS_NOTMASTER
A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the **/var/nis/NIS_SHARED_DIRCACHE** file will need to have their cache managers restarted (use **nis_cachemgr -i** to flush this cache).

NIS_NOTSAMEOBJ
An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOTSEARCHABLE
The table name resolved to a NIS+ object that was not searchable.

NIS_PARTIAL
This result is similar to **NIS_NOTFOUND** except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy.

NIS_RPCERROR
This fatal error indicates the RPC subsystem failed in some way. Generally there will be a **syslog**(3) message indicating why the RPC request failed.

NIS_S_NOTFOUND
The named entry does not exist in the table; however, not all tables in the path could be searched, so the entry may exist in one of those tables.

NIS_S_SUCCESS
Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible.

NIS_SUCCESS
The request was successful.

NIS_SYSTEMERROR
Some form of generic system error occurred while attempting the request. Check the **syslog**(3) record for error messages from the server.

NIS_TOOMANYATTRS
The search criteria passed to the server had more attributes than the table had searchable columns.

NIS_TRYAGAIN
The server connected to was too busy to handle your request. **add_entry( )**, **remove_entry( )**, and **modify_entry( )** return this error when the master server is currently updating its internal state. It can be returned to **nis_list( )** when the function specifies a callback and the server does not have the resources to handle callbacks.

NIS_TYPEMISMATCH
An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table.

Summary of Trusted

To succeed, **nis_modify_entry( )** must inherit the **PAF_TRUSTED_PATH** attribute.

nis_next_entry (NIS+ API)

Purpose

Used to fetch entries from a table one at a time.

Syntax

cc [***flag*** . . .] ***file***. . . **-lnsl** [***library. . .***]

```
#include <rpcsvc/nis.h>
```

nis_result * nis_next_entry(nis_name *table_name*, netobj *cookie*)

Description

One of a group of NIS+ APIs that is used to search and modify NIS+ tables, **nis_next_entry( )** is used to retrieve the "next" entry from a table specified by ***table_name***.

Entries within a table are named by NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket **[ ]** characters. Indexed names have the following form:

```
[ colname=value,...],tablename
```

nis_next_entry( ) retrieves the "next" entry from a table specified by ***table_name***. The order in which entries are returned is not guaranteed. Further, should an update occur in the table between client calls to **nis_next_entry( )**, there is no guarantee that an entry that is added or modified will be seen by the client. Should an entry be removed from the table that would have been the "next" entry returned, the error **NIS_CHAINBROKEN** is returned instead.

Return Values

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
           nis_error status;
           struct {
                      u_int        objects_len;
                      nis_object   * objects_val;
           } objects;
           netobj     cookie;
           u_long     zticks;
           u_long     dticks;
           u_long     aticks;
           u_long     cticks;
     };
```

The ***status*** member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno( )**.

The **objects** structure contains two members: ***objects_val*** is an array of ***nis_object*** structures; ***objects_len*** is the number of cells in the array. These objects will be freed by a call to **nis_freeresult( )**. If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object( )** and freed with the function **nis_destroy_object( )**.

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

zticks
: The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks
: The time spent in the database backend, this time is measured from the time a database call starts, until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks
The time spent in any *accelerators* or caches. This includes the time required to locate the server needed to resolve the request.

cticks
The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating an NIS+ request.

Subtracting the value in ***dticks*** from the value in ***zticks*** will yield the time spent in the service code itself. Subtracting the sum of the values in ***zticks*** and ***aticks*** from the value in ***cticks*** will yield the time spent in the client library itself.

Note: All of the tick times are measured in microseconds.

Errors

The client library can return a variety of error returns and diagnostics. Following are some of the more pertient ones:

NIS_BADATTRIBUTE
The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value.

NIS_BADNAME
The name passed to the function is not a legal NIS+ name.

NIS_BADREQUEST
A problem was detected in the request structure passed to the client library.

NIS_CACHEEXPIRED
The entry returned came from an object cache that has ***expired***. This means that the time to live value has gone to zero and the entry may have changed. If the flag **NO_CACHE** was passed to the lookup function, the lookup function will retry the operation to get an unexpired copy of the object.

NIS_CBERROR
An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded.

NIS_CBRESULTS
Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result.

NIS_FOREIGNNS
The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ
The object pointed to by ***object*** is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL
The attempted modification failed.

NIS_NAMEEXISTS
An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOCALLBACK
The server was unable to contact the callback service on your machine. This results in no data being returned.

NIS_NOMEMORY
Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME
This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table does not know about the directory in which the table resides.

NIS_NOSUCHTABLE
The named table does not exist.

NIS_NOT_ME
A request was made to a server that does not serve the given name. Normally, this will not occur; however, if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND
No entries in the table matched the search criteria. If the search criteria was null (return all entries), then this result means that the table is empty and may safely be removed by calling the **nis_remove( )**. If the **FOLLOW_PATH** flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria.

NIS_NOTMASTER
A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the **/var/nis/NIS_SHARED_DIRCACHE** file will need to have their cache managers restarted (use **nis_cachemgr -i** to flush this cache).

NIS_NOTSAMEOBJ
An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOTSEARCHABLE
The table name resolved to a NIS+ object that was not searchable.

NIS_PARTIAL
This result is similar to **NIS_NOTFOUND**, except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy.

NIS_RPCERROR
This fatal error indicates the RPC subsystem failed in some way. Generally there will be a **syslog**(3) message indicating why the RPC request failed.

NIS_S_NOTFOUND
The named entry does not exist in the table; however, not all tables in the path could be searched, so the entry may exist in one of those tables.

NIS_S_SUCCESS
Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible.

NIS_SUCCESS
The request was successful.

NIS_SYSTEMERROR
Some form of generic system error occurred while attempting the request. Check the **syslog**(3) record for error messages from the server.

NIS_TOOMANYATTRS
The search criteria passed to the server had more attributes than the table had searchable columns.

NIS_TRYAGAIN
The server connected to was too busy to handle your request. **add_entry( )**, **remove_entry( )**, and **modify_entry( )** return this error when the master server is currently updating its internal state. It can be returned to **nis_list( )** when the function specifies a callback and the server does not have the resources to handle callbacks.

NIS_TYPEMISMATCH
An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table.

nis_perror (NIS+ API)

Purpose

Prints the error message corresponding to ***status*** as `"label: error message"` on standard error.

Syntax

```
cc
 [
flag
 ... ]
file
 ...
-lnsl
 [
library
 ... ]
#include <rpcsvc/nis.h>
```

```
char *  nis_sperrno(nis_error status);
void nis_perror(nis_error status, char * label);
void nis_lerror(nis_error status, char * label);
char * nis_sperror_r(nis_error status, char * label, char * buf, int length);
char * nis_sperror(nis_error status, char * label);
```

Description

One of a group of NIS+ APIs that convert NIS+ status values into strings, **nis_perror** prints the error messages corresponding to *status* as `"label: error messages"` on standard error.

nis_remove_entry (NIS+ API)

Purpose

Used to remove an NIS+ object from the NIS+ table_name.

Syntax

cc [***flag*** . . .] ***file***. . . **-lnsl** [***library. . .***]

```
#include <rpcsvc/nis.h>
```

nis_result * nis_remove_entry(nis_name *name*, nis_object, * *object*, u_long *flags*);

Description

One of a group of NIS+ APIs that is used to search and modify NIS+ tables, **nis_remove_entry( )** is used to remove the identified entry from the table or a set of entries identified by ***table_name***.

Entries within a table are named by NIS+ indexed names. An indexed name is a compound name that is composed of a search criteria and a simple NIS+ name that identifies a table object. A search criteria is a series of column names and their associated values enclosed in bracket **[ ]** characters. Indexed names have the following form:

```
[ colname=value,...],tablename
```

nis_remove_entry( ) removes the identified entry from the table or a set of entries identified by ***table_name***. If the parameter ***object*** is non-null, it is presumed to point to a cached copy of the entry. When the removal is attempted, and the object that would be removed is not the same as the cached object pointed to by ***object***, then the operation will fail with an **NIS_NOTSAMEOBJ** error. If an object is passed with this function, the search criteria in name is optional as it can be constructed from the values within the entry. However, if no object is present, the search criteria must be included in the ***name*** parameter. If the flags variable is null, and the search criteria does not uniquely identify an entry, the **NIS_NOTUNIQUE** error is returned and the operation is aborted. If the flag parameter **REM_MULTIPLE** is passed, and if remove permission is allowed for each of these objects, then all objects that match the search criteria will be removed. Note that a null search criteria and the **REM_MULTIPLE** flag will remove all entries in a table.

To succeed, **nis_remove_entry( )** must inherit the **PAF_TRUSTED_PATH** attribute.

Return Values

These functions return a pointer to a structure of type **nis_result**:

```
struct nis_result {
           nis_error status;
           struct {
                      u_int        objects_len;
                      nis_object   * objects_val;
           } objects;
           netobj     cookie;
           u_long     zticks;
           u_long     dticks;
           u_long     aticks;
           u_long     cticks;
     };
```

The ***status*** member contains the error status of the the operation. A text message that describes the error can be obtained by calling the function **nis_sperrno( )**.

The **objects** structure contains two members: ***objects_val*** is an array of ***nis_object*** structures; ***objects_len*** is the number of cells in the array. These objects will be freed by a call to **nis_freeresult( )**. If you need to keep a copy of one or more objects, they can be copied with the function **nis_clone_object( )** and freed with the function **nis_destroy_object( )**.

The various ticks contain details of where the time (in microseconds) was taken during a request. They can be used to tune one's data organization for faster access and to compare different database implementations.

zticks
The time spent in the NIS+ service itself, this count starts when the server receives the request and stops when it sends the reply.

dticks
The time spent in the database backend, this time is measured from the time a database call starts until a result is returned. If the request results in multiple calls to the database, this is the sum of all the time spent in those calls.

aticks
The time spent in any *accelerators* or caches. This includes the time required to locate the server needed to resolve the request.

cticks
The total time spent in the request, this clock starts when you enter the client library and stops when a result is returned. By subtracting the sum of the other ticks values from this value you can obtain the local overhead of generating an NIS+ request.

Subtracting the value in ***dticks*** from the value in ***zticks*** will yield the time spent in the service code itself. Subtracting the sum of the values in ***zticks*** and ***aticks*** from the value in ***cticks*** will yield the time spent in the client library itself.

Note: All of the tick times are measured in microseconds.

Errors

The client library can return a variety of error returns and diagnostics. Following are some of the mor pertient ones:

NIS_BADATTRIBUTE
The name of an attribute did not match up with a named column in the table, or the attribute did not have an associated value.

NIS_BADNAME
The name passed to the function is not a legal NIS+ name.

NIS_BADREQUEST
A problem was detected in the request structure passed to the client library.

NIS_CACHEEXPIRED
The entry returned came from an object cache that has ***expired***. This means that the time to live value has gone to zero and the entry may have changed. If the flag **NO_CACHE** was passed to the lookup function, the lookup function will retry the operation to get an unexpired copy of the object.

NIS_CBERROR
An RPC error occurred on the server while it was calling back to the client. The transaction was aborted at that time and any unsent data was discarded.

NIS_CBRESULTS
Even though the request was successful, all of the entries have been sent to your callback function and are thus not included in this result.

NIS_FOREIGNNS
The name could not be completely resolved. When the name passed to the function would resolve in a namespace that is outside the NIS+ name tree, this error is returned with a NIS+ object of type **DIRECTORY**. The returned object contains the type of namespace and contact information for a server within that namespace.

NIS_INVALIDOBJ
The object pointed to by ***object*** is not a valid NIS+ entry object for the given table. This could occur if it had a mismatched number of columns, or a different data type (for example, binary or text) than the associated column in the table.

NIS_LINKNAMEERROR
The name passed resolved to a **LINK** type object and the contents of the object pointed to an invalid name.

NIS_MODFAIL
The attempted modification failed.

NIS_NAMEEXISTS
An attempt was made to add a name that already exists. To add the name, first remove the existing name and then add the new name or modify the existing named object.

NIS_NAMEUNREACHABLE
This soft error indicates that a server for the desired directory of the named table object could not be reached. This can occur when there is a network partition or the server has crashed. Attempting the operation again may succeed. See the **HARD_LOOKUP** flag.

NIS_NOCALLBACK
The server was unable to contact the callback service on your machine. This results in no data being returned.

NIS_NOMEMORY
Generally a fatal result. It means that the service ran out of heap space.

NIS_NOSUCHNAME
This hard error indicates that the named directory of the table object does not exist. This occurs when the server that should be the parent of the server that serves the table does not know about the directory in which the table resides.

NIS_NOSUCHTABLE
The named table does not exist.

NIS_NOT_ME
A request was made to a server that does not serve the given name. Normally, this will not occur; however, if you are not using the built in location mechanism for servers, you may see this if your mechanism is broken.

NIS_NOTFOUND
No entries in the table matched the search criteria. If the search criteria was null (return all entries), then this result means that the table is empty and may safely be removed by calling the **nis_remove( )**. If the **FOLLOW_PATH** flag was set, this error indicates that none of the tables in the path contain entries that match the search criteria.

NIS_NOTMASTER
A change request was made to a server that serves the name, but it is not the master server. This can occur when a directory object changes and it specifies a new master server. Clients that have cached copies of the directory object in the **/var/nis/NIS_SHARED_DIRCACHE** file will need to have their cache managers restarted (use **nis_cachemgr -i** to flush this cache).

NIS_NOTSAMEOBJ
An attempt to remove an object from the namespace was aborted because the object that would have been removed was not the same object that was passed in the request.

NIS_NOTSEARCHABLE
The table name resolved to a NIS+ object that was not searchable.

NIS_PARTIAL
This result is similar to **NIS_NOTFOUND** except that it means the request succeeded but resolved to zero entries. When this occurs, the server returns a copy of the table object instead of an entry so that the client may then process the path or implement some other local policy.

NIS_RPCERROR
This fatal error indicates the RPC subsystem failed in some way. Generally there will be a **syslog**(3) message indicating why the RPC request failed.

NIS_S_NOTFOUND
The named entry does not exist in the table; however, not all tables in the path could be searched, so the entry may exist in one of those tables.

NIS_S_SUCCESS
Even though the request was successful, a table in the search path was not able to be searched, so the result may not be the same as the one you would have received if that table had been accessible.

NIS_SUCCESS
The request was successful.

NIS_SYSTEMERROR
Some form of generic system error occurred while attempting the request. Check the **syslog**(3) record for error messages from the server.

NIS_TOOMANYATTRS
The search criteria passed to the server had more attributes than the table had searchable columns.

NIS_TRYAGAIN
The server connected to was too busy to handle your request. **add_entry( )**, **remove_entry( )**, and **modify_entry( )** return this error when the master server is currently updating its internal state. It can be returned to **nis_list( )** when the function specifies a callback and the server does not have the resources to handle callbacks.

NIS_TYPEMISMATCH
An attempt was made to add or modify an entry in a table, and the entry passed was of a different type than the table.

Summary of Trusted

To succeed, **nis_remove_entry( )** must inherit the **PAF_TRUSTED_PATH** attribute.

nis_sperror (NIS+ API)

Purpose

Returns a pointer to a string that can be used or copied using the **strdup** function.

Syntax

```
cc
 [
flag
 ... ]
file
 ...
-lnsl
 [
library
 ... ]
#include <rpcsvc/nis.h>
```

```
char * nis_sperror(nis_error status, char * label);
```

Description

One of a group of NIS+ APIs that convert NIS+ status values into strings, **nis_sperror** returns a pointer to a string that can be used or copied using the **strdup** function. The caller must supply a string buffer, ***buf***, large enough to hold the error string (a buffer size of 128 bytes is guaranteed to be sufficiently large). ***status*** and ***label*** are the same as for **nis_perror**. The pointer returned by the function is a pointer to ***buf***. ***length*** specifies the number of characters to copy from the error string to ***buf***. The string is returned as a pointer to a buffer that is reused on each call.

Note: When compiling multithreaded applications, see *Writing Reentrant and Thread-Safe Code* for information about the use of the **_REENTRANT** flag.

Simple Network Management Protocol (SNMP)

The Simple Network Management Protocol (SNMP) is used by network hosts to exchange information in the management of networks. SNMP network management is based on the familiar client-server model that is widely used in Transmission Control Protocol/Internet Protocol (TCP/IP)-based network applications. Each managed host runs a process called an agent. The agent is a server process that maintains the MIB database for the host. Hosts that are involved in network management decision-making may run a process called a manager. A manager is a client application that generates requests for MIB information and processes responses. In addition, a manager may send requests to agent servers to modify MIB information.

getsmuxEntrybyname or getsmuxEntrybyidentity Subroutine

Purpose

Retrieves SNMP multiplexing (SMUX) peer entries from the **/etc/snmpd.peers** file or the local **snmpd.peers** file.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>

struct smuxEntry *getsmuxEntrybyname ( name)
char *name;

struct smuxEntry *getsmuxEntrybyidentity ( identity)
OID identity;
```

Description

The **getsmuxEntrybyname** and **getsmuxEntrybyidentity** subroutines read the **snmpd.peers** file and retrieve information about the SMUX peer. The sample peers file is **/etc/snmpd.peers**. However, these subroutines can also retrieve the information from a copy of the file that is kept in the local directory. The **snmpd.peers** file contains entries for the SMUX peers defined for the network. Each SMUX peer entry should contain:

- The name of the SMUX peer.
- The SMUX peer object identifier.
- An optional password to be used on connection initiation. The default password is a null string.
- The optional priority to register the SMUX peer. The default priority is 0.

The **getsmuxEntrybyname** subroutine searches the file for the specified name. The **getsmuxEntrybyidentity** subroutine searches the file for the specified object identifier.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
name	Points to a character string that names the SMUX peer.
identity	Specifies the object identifier for a SMUX peer.

Return Values

If either subroutine finds the specified SMUX entry, that subroutine returns a structure containing the entry. Otherwise, a null entry is returned.

Files

Item	Description
/etc/snmpd.peers	Contains the SMUX peer definitions for the network.

Related information
List of Network Manager Programming References
SNMP Overview for Programmers

isodetailor Subroutine

Purpose

Initializes variables for various logging facilities.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <isode/tailor.h>

void isodetailor (myname, wantuser)
char * myname;
int  wantuser;
```

Description

The ISODE library contains internal logging facilities. Some of the facilities need to have their variables initialized. The **isodetailor** subroutine sets default or user-defined values for the logging facility variables. The logging facility variables are listed in the **usr/lpp/snmpd/smux/isodetailor** file.

The **isodetailor** subroutine first reads the **/etc/isodetailor** file. If the *wantuser* parameter is set to 0, the **isodetailor** subroutine ignores the *myname* parameter and reads the **/etc/isodetailor** file. If the *wantuser* parameter is set to a value greater than 0, the **isodetailor** subroutine searches the current user's home directory (**$HOME**) and reads a file based on the *myname* parameter. If the *myname* parameter is specified, the **isodetailor** subroutine reads a file with the name in the form **.***myname***_tailor**. If the *myname* parameter is null, the **isodetailor** subroutine reads a file named **.isode_tailor**. The **_tailor** file contents must be in the following form:

```
#comment
<variable> : <value>  # comment
<variable> : <value>  # comment
<variable> : <value>  # comment
```

The comments are optional. The **isodetailor** subroutine reads the file and changes the values. The latest entry encountered is the final value. The subroutine reads **/etc/isodetailor** first and then the **$HOME** directory, if told to do so. A complete list of the variables is in the **/usr/lpp/snmpd/smux/isodetailor** sample file.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
myname	Contains a character string describing the SNMP multiplexing (SMUX) peer.
wantuser	Indicates that the **isodetailor** subroutine should check the **$HOME** directory for a **isodetailor** file if the value is greater than 0. If the value of the *wantuser* parameter is set to 0, the **$HOME** directory is not checked, and the *myname* parameter is ignored.

Files

Item	Description
/etc/isodetailor	Location of user's copy of the **/usr/lpp/snmpd/smux/isodetailor** file.
/usr/lpp/snmpd/smux/isodetailor	Contains a complete list of all the logging parameters.

Related reference
ll_hdinit, ll_dbinit, _ll_log, or ll_log Subroutine
Related information
List of Network Manager Programming References
SNMP Overview for Programmers

ll_hdinit, ll_dbinit, _ll_log, or ll_log Subroutine

Purpose

Reports errors to log files.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <isode/logger.h>

void ll_hdinit (lp, prefix)
register LLog * lp;
char * prefix;

void ll_dbinit (lp, prefix)
register LLog *lp;
char *prefix;

int _ll_log (lp, event, ap)
register LLog *lp;
int  event;
va_list  ap;

int ll_log ( va_alist)
va_dcl
```

Description

The ISODE library provides logging subroutines to put information into log files. The **LLog** data structure contains the log file information needed to control the associated log. The SMUX peer provides the log file information to the subroutines.

The **LLog** structure contains the following fields:

```
typedef struct ll_struct
{
char      *ll_file;    /* path name to logging file         */
char      *ll_hdr;     /* text to put in opening line       */
char      *ll_dhdr;    /* dynamic header - changes          */
int       ll_events;   /* loggable events                   */
```

```
    int       ll_syslog;   /* loggable events to send to syslog */
                           /* takes same values as ll_events    */
    int       ll_msize;    /* max size for log, in Kbytes       */
                           /* If ll_msize < 0, then no checking */
    int       ll_stat;     /* assorted switches                 */
    int       ll_fd;       /* file descriptor                   */
    } LLog;
```

The possible values for the `ll_events` and `ll_syslog` fields are:

```
    LLOG_NONE         0         /*  No logging is performed       */
    LLOG_FATAL        0x01      /*  fatal errors                  */
    LLOG_EXCEPTIONS   0x02      /*  exceptional events            */
    LLOG_NOTICE       0x04      /*  informational notices         */
    LLOG_PDUS         0x08      /*  PDU printing                  */
    LLOG_TRACE        0x10      /*  program tracing               */
    LLOG_DEBUG        0x20      /*  full debugging                */
    LLOG_ALL          0xff      /*  All of the above logging      */
```

The possible values for the `ll_stat` field are:

```
   LLOGNIL           0x00      /* No status information           */
   LLOGCLS           0x01      /* keep log closed, except writing */
   LLOGCRT           0x02      /* create log if necessary         */
   LLOGZER           0x04      /* truncate log when limits reach  */
   LLOGERR           0x08      /* log closed due to (soft) error  */
   LLOGTTY           0x10      /* also log to stderr              */
   LLOGHDR           0x20      /* static header allocated/filled  */
   LLOGDHR           0x40      /* dynamic header allocated/filled */
```

The **ll_hdinit** subroutine fills the `ll_hdr` field of the **LLog** record. The subroutine allocates the memory of the static header and creates a string with the information specified by the *prefix* parameter, the current user's name, and the process ID of the SMUX peer. It also sets the static header flag in the `ll_stat` field. If the *prefix* parameter value is null, the header flag is set to the "unknown" string.

The **ll_dbinit** subroutine fills the `ll_file` field of the **LLog** record. If the *prefix* parameter is null, the `ll_file` field is not changed. The **ll_dbinit** subroutine also calls the **ll_hdinit** subroutine with the same *lp* and *prefix* parameters. The **ll_dbinit** subroutine sets the log messages to **stderr** and starts the logging facility at its highest level.

The **_ll_log** and **ll_log** subroutines are used to print to the log file. When the **LLog** structure for the log file is set up, the **_ll_log** or **ll_log** subroutine prints the contents of the string format, with all variables filled in, to the log specified in the *lp* parameter. The **LLog** structure passes the name of the target log to the subroutine.

The expected parameter format for the **_ll_log** and **ll_log** subroutines is:

- **_ll_log(***lp***,** *event***,** *what***),** *string_format***,** ...**);**
- **ll_log(***lp***,** *event***,** *what***,** *string_format***,** ...**);**

The difference between the **_ll_log** and the **ll_log** subroutine is that the **_ll_log** uses an explicit listing of the **LLog** structure and the *event* parameter. The **ll_log** subroutine handles all the variables as a variable list.

The *event* parameter specifies the type of message being logged. This value is checked against the `events` field in the log record. If it is a valid event for the log, the other **LLog** structure variables are written to the log.

The *what* parameter variable is a string that explains what actions the subroutines have accomplished. The rest of the variables should be in the form of a **printf** statement, a string format and the variables to fill the various variable placeholders in the string format. The final output of the logging subroutine is in the following format:

```
mm/dd hh:mm:ss ll_hdr ll_dhdr string_format what: system_error
```

where:

Variable	Description
`mm/dd`	Specifies the date.
`hh:mm:ss`	Specifies the time.
`ll_hdr`	Specifies the value of the `ll_hdr` field of the **LLog** structure.
`ll_dhdr`	Specifies the value of the `ll_dhdr` field of the **LLog** structure.
`string_format`	Specifies the string format passed to the **ll_log** subroutine, with the extra variables filled in.
`what`	Specifies the variable that tells what has occurred. The `what` variable often contains the reason for the failure. For example if the memory device, **/dev/mem**, fails, the `what` variable contains the name of the **/dev/mem** device.
`system_error`	Contains the string for the **errno** value, if it exists.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
lp	Contains a pointer to a structure that describes a log file. The *lp* parameter is used to describe things entered into the log, the file name, and headers.
prefix	Contains a character string that is used to represent the name of the SMUX peer in the **ll_hdinit** subroutine. In the **ll_dbinit** subroutine, the *prefix* parameter represents the name of the log file to be used. The new log file name will be **./***prefix***.log**.
event	Specifies the type of message to be logged.
ap	Provides a list of variables that is used to print additional information about the status of the logging process. The first argument needs to be a character string that describes what failed. The following arguments are expected in a format similar to the **printf** operation, which is a string format with the variables needed to fill the format variable places.
va_alist	Provides a variable list of parameters that includes the *lp*, *event*, and *ap* variables.

Return Values

The **ll_dbinit** and **ll_hdinit** subroutines have no return values. The **_ll_log** and **ll_log** subroutines return **OK** on success and **NOTOK** on failure.

Related reference
isodetailor Subroutine
Related information
List of Network Manager Programming References
Examples of SMUX Error Logging Routines
SNMP Overview for Programmers

o_number, o_integer, o_string, o_igeneric, o_generic, o_specific, or o_ipaddr Subroutine

Purpose

Encodes values retrieved from the Management Information Base (MIB) into the specified variable binding.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/objects.h>
#include <isode/pepsy/SNMP-types.h>
#include <sys/types.h>
#include <netinet/in.h>
```

int o_number (*oi*, *v*, *number*)
OI *oi*;
register struct type_SNMP_VarBind ***v**;
int *number*;

```
#define o_integer (oi, v, number) o_number ((oi), (v), (number))
```

int o_string (*oi*, *v*, *base*, *len*)
OI *oi*;
register struct type_SNMP_VarBind ***v**;
char ***base**;
int *len*;

int o_igeneric (*oi*, *v*, *offset*)
OI *oi*;
register struct type_SNMP_VarBind ***v**;
int *offset*;

```
int o_generic (oi, v, offset)
OI oi;
register struct type_SNMP_VarBind *v;
int offset;
```

int o_specific (*oi*, *v*, *value*)
OI *oi*;
register struct type_SNMP_VarBind ***v**;
caddr_t *value*;

int o_ipaddr (*oi*, *v*, *netaddr*)
OI *oi*;
register struct type_SNMP_VarBind ***v**;
struct sockaddr_in ***netaddr**;

Description

The **o_number** subroutine assigns a number retrieved from the MIB to the variable binding used to request it. Once an MIB value has been retrieved, the value must be stored in the binding structure associated with the variable requested. The **o_number** subroutine places the integer *number* into the *v* parameter, which designates the binding for the variable. The *value* parameter type is defined by the *oi* parameter and is used to specify the encoding subroutine that stores the value. The *oi* parameter references a specific MIB variable and should be the same as the variable specified in the *v* parameter. The encoding functions are defined for each type of variable and are contained in the object identifier (**OI**) structure.

The **o_integer** macro is defined in the **/usr/include/snmp/objects.h** file. This macro casts the *number* parameter as an integer. Use the **o_integer** macro for types that are not integers but have integer values.

The **o_string** subroutine assigns a string that has been retrieved for a MIB variable to the variable binding used to request the string. Once a MIB variable has been retrieved, the value is stored in the binding structure associated with the variable requested. The **o_string** subroutine places the string, specified with

the *base* parameter, into the variable binding in the *v* parameter. The length of the string represented in the *base* parameter equals the value of the *len* parameter. The length is used to define how much of the string is copied in the binding parameter of the variable. The *value* parameter type is defined by the *oi* parameter and is used to specify the encoding subroutine that stores the value. The *oi* parameter references a specific MIB variable and should be the same as the variable specified in the *v* parameter. The encoding subroutines are defined for each type of variable and are contained in the **OI** structure.

The **o_generic** and **o_igeneric** subroutines assign results that are already in the customer's MIB database. These two subroutines do not retrieve values from any other source. These subroutines check whether the MIB database has information on how and what to encode as the value. The **o_generic** and **o_igeneric** subroutines also ensure that the variable requested is an instance. If the variable is an instance, the subroutines encode the value and return **OK**. The subroutine has an added set of return codes. If there is not any information about the variable, the subroutine returns **NOTOK** on a **get_next** request and **int_SNMP_error__status_noSuchName** for the get and set requests. The difference between the **o_generic** and the **o_igeneric** subroutine is that the **o_igeneric** subroutine provides a method for users to define a generic subroutine.

The **o_specific** subroutine sets the binding value for a MIB variable with the value in a character pointer. The **o_specific** subroutine ensures that the data-encoding procedure is defined. The encode subroutine is always checked by all of the **o_** subroutines. The **o_specific** subroutine returns the normal values.

The **o_ipaddr** subroutine sets the binding value for variables that are network addresses. The **o_ipaddr** subroutine uses the `sin_addr` field of the **sockaddr_in** structure to get the address. The subroutine does the normal checking and returns the results like the rest of the subroutines.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
oi	Contains the **OI** data structure for the variable whose value is to be recorded into the binding structure.
v	Specifies the variable binding parameter, which is of type **type_SNMP_VarBind**. The *v* parameter contains a `name` and a `value` field. The `value` field contents are supplied by the **o_** subroutines.
number	Contains an integer to store in the `value` field of the *v* (variable bind) parameter.
base	Points to the character string to store in the `value` field of the *v* parameter.
len	Designates the length of the integer character string to copy. The character string is described by the *base* parameter.
offset	Contains an integer value of the current type of request, for example: `type_SNMP_PDUs_get__next__request`
value	Contains a character pointer to a value.
netaddr	Points to a **sockaddr_in** structure. The subroutine only uses the `sin_addr` field of this structure.

Return Values

The return values for these subroutines are:

Value	Description
int_SNMP_error__status_genErr	Indicates an error occurred when setting the *v* parameter value.
int_SNMP_error__status_noErr	Indicates no errors found.

Related reference
s_generic Subroutine
Related information
List of Network Manager Programming References
SNMP Overview for Programmers
Working with Management Information Base (MIB) Variables

oid_cmp, oid_cpy, oid_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode_aux, prim2oid, or oid2prim Subroutine

Purpose

Manipulates the object identifier data structure.

Library

ISODE Library (**libisode.a**)

Syntax

```
#include <isode/psap.h>
```

```
int oid_cmp (p, q)
OID  p,  q;

OID oid_cpy (oid)
OID  oid;
```

```
void oid_free (oid)
OID oid;
```

```
char *sprintoid (oid)
OID oid;
```

```
OID str2oid (s)
char * s;

OID ode2oid (descriptor)
char * descriptor;
```

```
char *oid2ode (oid)
OID oid;
```

```
OID *oid2ode_aux (descriptor, quote)
char *descriptor;
int  quote;

OID prim2oid (pe)
PE  pe;
```

```
PE oid2prim (oid)
OID oid;
```

Description

These subroutines are used to manipulate and translate object identifiers. The object identifier data (**OID**) structure and these subroutines are defined in the **/usr/include/isode/psap.h** file.

The **oid_cmp** subroutine compares two **OID** structures. The **oid_cpy** subroutine copies the object identifier, specified by the *oid* parameter, into a new structure. The **oid_free** procedure frees the object identifier and does not have any return parameters.

The **sprintoid** subroutine takes an object identifier and returns the dot-notation description as a string. The string is in static storage and must be copied to other user storage if it is to be maintained. The **sprintoid** subroutine takes the object data and converts it without checking for the existence of the *oid* parameter.

The **str2oid** subroutine takes a character string specifying an object identifier in dot notation (for example, `1.2.3.6.1.2`) and converts it into an **OID** structure. The space is static. To get a permanent copy of the **OID** structure, use the **oid_cpy** subroutine.

The **oid2ode** subroutine is identical to the **sprintoid** subroutine except that the **oid2ode** subroutine checks whether the *oid* parameter is in the **isobjects** database. The **oid2ode** subroutine is implemented as a macro call to the **oid2ode_aux** subroutine. The **oid2ode_aux** subroutine is similar to the **oid2ode** subroutine except for an additional integer parameter that specifies whether the string should be enclosed by quotes. The **oid2ode** subroutine always encloses the string in quotes.

The **ode2oid** subroutine retrieves an object identifier from the **isobjects** database.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
p	Specifies an **OID** structure.
q	Specifies an **OID** structure.
descriptor	Contains the object identifier descriptor data.
oid	Contains the object identifier data.
s	Contains a character string that defines an object identifier in dot notation.
descriptor	Contains the object identifier descriptor data.
quote	Specifies an integer that indicates whether a string should be enclosed in quotes. A value of 1 adds quotes; a value of 0 does not add quotes.
pe	Contains a presentation element in which the **OID** structure is encoded (as with the **oid2prim** subroutine) or decoded (as with the **prim2oid** subroutine).

Return Values

The **oid_cmp** subroutine returns a 0 if the structures are identical, -1 if the first object is less than the second, and a 1 if any other conditions are found. The **oid_cpy** subroutine returns a pointer to the designated object identifier when the subroutine is successful.

The **oid2ode** subroutine returns the dot-notation description as a string in quotes. The **sprintoid** subroutine returns the dot-notation description as a string without quotes.

The **ode2oid** subroutine returns a static pointer to the object identifier. If the **ode2oid** and **oid_cpy** subroutines are not successful, the **NULLOID** value is returned.

Related reference

oid_extend or oid_normalize Subroutine

text2oid or text2obj Subroutine

Related information

List of Network Manager Programming References

SNMP Overview for Programmers

oid_extend or oid_normalize Subroutine

Purpose

Extends the base ISODE library subroutines.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/objects.h>

OID oid_extend (q, howmuch)
OID  q;
integer  howmuch;

OID oid_normalize (q, howmuch, initial)
OID q;
integer howmuch,  initial;
```

Description

The **oid_extend** subroutine is used to extend the current object identifier data (**OID**) structure. The **OID** structure contains an integer number of entries and an array of integers. The **oid_extend** subroutine creates a new, extended **OID** structure with an array of the size specified in the *howmuch* parameter plus the original array size specified in the *q* parameter. The original values are copied into the first entries of the new structure. The new values are uninitialized. The entries of the **OID** structure are used to represent the values of an Management Information Base (MIB) tree in dot notation. Each entry represents a level in the MIB tree.

The **oid_normalize** subroutine extends and adjusts the values of the **OID** structure entries. The **oid_normalize** subroutine extends the **OID** structure and then decrements all nonzero values by 1. The new values are initialized to the value of the *initial* parameter. This subroutine stores network address and netmask information in the **OID** structure.

These subroutines do not free the *q* parameter.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
q	Specifies the size of the original array.
howmuch	Specifies the size of the array for the new **OID** structure.
initial	Indicates the initialized value of the **OID** structure extensions.

Return Values

Both subroutines, when successful, return the pointer to the new object identifier structure. If the subroutines fail, the **NULLOID** value is returned.

Related reference

oid_cmp, oid_cpy, oid_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode_aux, prim2oid, or oid2prim Subroutine

Related information
List of Network Manager Programming References
SNMP Overview for Programmers

readobjects Subroutine

Purpose

Allows the SNMP multiplexing (SMUX) peer to read the Management Information Base (MIB) variable structure.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/objects.h>
```

```
int
readobjects ( file)
char *file;
```

Description

The **readobjects** subroutine reads the file given in the *file* parameter. This file must contain the MIB variable descriptions that the SMUX peer supports. The SNMP library functions require basic information about the MIB tree supported by the SMUX peer. These structures are supplied from information in the **readobjects** file. The **text2oid** subroutine receives a string description and uses the object identifier information retrieved with the **readobjects** subroutine to return a MIB object identifier. The file designated in the *file* parameter must be in the following form:

```
<MIB directory> <MIB position>

<MIB name> <MIB position> <MIB type> <MIB access> <MIB required?>
<MIB name> <MIB position> <MIB type> <MIB access> <MIB required?>
...
```

An example of a file that uses this format is **/etc/mib.defs**. The **/etc/mib.defs** file defines the MIBII tree used in the SNMP agent.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
file	Contains the name of the file to be read. If the value is NULL, the **/etc/mib.defs** file is read.

Return Values

If the subroutine is successful, **OK** is returned. Otherwise, **NOTOK** is returned.

Related reference
text2oid or text2obj Subroutine
smux_free_tree Subroutine
Related information
List of Network Manager Programming References

SNMP Overview for Programmers

s_generic Subroutine

Purpose

Sets the value of the Management Information Base (MIB) variable in the database.

Library

The SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/objects.h>
```

```
int s_generic
( oi,  v,  offset)
OI oi;
register struct type_SNMP_VarBind *v;
int offset;
```

Description

The **s_generic** subroutine sets the database value of the MIB variable. The subroutine retrieves the information it needs from a value in a variable binding within the Protocol Data Unit (PDU). The **s_generic** subroutine sets the MIB variable, specified by the object identifier *oi* parameter, to the `value` field specified by the *v* parameter.

The *offset* parameter is used to determine the stage of the set process. If the *offset* parameter value is **type_SNMP_PDUs_set__reque st**, the value is checked for validity and the value in the `ot_save` field in the **OI** structure is set. If the *offset* parameter value is **type_SNMP_PDUs_commit**, the value in the `ot_save` field is freed and moved to the MIB `ot_info` field. If the *offset* parameter value is **type_SNMP_PDUs_rollback**, the value in the `ot_save` field is freed and no new value is written.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
oi	Designates the **OI** structure representing the MIB variable to be set.
v	Specifies the variable binding that contains the value to be set.
offset	Contains the stage of the set. The possible values for the *offset* parameter are **type_SNMP_PDUs_commit**, **type_SNMP_PDUs_rollback**, or **type_SNMP_PDUs_set__request**.

Return Values

If the subroutine is successful, a value of **int_SNMP_error__status_noError** is returned. Otherwise, a value of **int_SNMP_error__status_badValue** is returned.

Related reference

o_number, o_integer, o_string, o_igeneric, o_generic, o_specific, or o_ipaddr Subroutine

Related information

List of Network Manager Programming References

SNMP Overview for Programmers

SNMP daemon processing

smux_close Subroutine

Purpose

Ends communications with the SNMP agent.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>
```

```
int smux_close ( reason)
int reason;
```

Description

The **smux_close** subroutine closes the transmission control protocol (TCP) connection from the SNMP multiplexing (SMUX) peer. The **smux_close** subroutine sends the close protocol data unit (PDU) with the error code set to the *reason* value. The subroutine closes the TCP connection and frees the socket. This subroutine also frees information it was maintaining for the connection.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
reason	Indicates an integer value denoting the reason the close PDU message is being sent.

Return Values

If the subroutine is successful, **OK** is returned. Otherwise, **NOTOK** is returned.

Error Codes

If the subroutine returns **NOTOK**, the **smux_errno** global variable is set to one of the following values:

Value	Description
invalidOperation	Indicates that the **smux_init** subroutine has not been executed successfully.
congestion	Indicates that memory could not be allocated for the close PDU. The TCP connection is closed.
youLoseBig	Indicates that the SNMP code has a problem. The TCP connection is closed.

Related information

List of Network Manager Programming References

SNMP Overview for Programmers

smux_error Subroutine

Purpose

Creates a readable string from the **smux_errno** global variable value.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>
```

char *smux_error (*error*)
int *error*;

Description

The **smux_error** subroutine creates a readable string from error code values in the **smux_errno** global variable in the **smux.h** file. The **smux** global variable, **smux_errno**, is set when an error occurs. The **smux_error** subroutine can also get a string that interprets the value of the **smux_errno** variable. The **smux_error** subroutine can be used to retrieve any numbers, but is most useful interpreting the integers returned in the **smux_errno** variable.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
error	Contains the error to interpret. Usually called with the value of the **smux_errno** variable, but can be called with any error that is an integer.

Return Values

If the subroutine is successful, a pointer to a static string is returned. If an error occurs, a string of the type SMUX `error %s(%d)` is returned. The %s value is a string representing the explanation of the error. The %d is the number used to reference that error.

Related information
List of Network Manager Programming References
SNMP Overview for Programmers

smux_free_tree Subroutine

Purpose

Frees the object tree when a **smux** tree is unregistered.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>
```

```
void smux_free_tree ( parent,  child)
char *parent;
char *child;
```

Description

The **smux_free_tree** subroutine frees elements in the Management Information Base (MIB) list within an SNMP multiplexing (SMUX) peer. If the SMUX peer implements the MIB list with the **readobjects** subroutine, a list of MIBs is created and maintained. These MIBs are kept in the object tree (**OT)** data structures.

Unlike the **smux_register** subroutine, the **smux_free_tree** subroutine frees the MIB elements even if the tree is unregistered by the **snmpd** daemon. This functionality is not performed by the **smux_register** routine because the **OT** list is created independently of registering a tree with the **snmpd** daemon. The unregistered objects should be removed as the user deems appropriate. Remove the unregistered objects if the **smux** peer is highly dynamic. If the peer registers and unregisters many trees, it might be reasonable to add and delete the **OT** MIB list on the fly. The **smux_free_tree** subroutine expects the parent of the MIB tree in the local **OT** list to delete unregistered objects.

This subroutine does not return values or error codes.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
parent	Contains a character string holding the immediate parent of the tree to be deleted.
child	Contains a character string holding the beginning of the tree to be deleted.

The character strings are names or dot notations representing object identifiers.

Related reference
readobjects Subroutine
Related information
snmpd subroutine
List of Network Manager Programming References
SNMP Overview for Programmers

smux_init Subroutine

Purpose

Initiates the transmission control protocol (TCP) socket that the SNMP multiplexing (SMUX) agent uses and clears the basic SMUX data structures.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>
```

```
int smux_init ( debug)
int debug;
```

Description

The **smux_init** subroutine initializes the TCP socket that is used by the SMUX agent to communicate with the SNMP daemon. The subroutine assumes that loopback is used to define the path to the SNMP daemon. Name resolution attempts to find an IPv6 address mapping for loopback. If it cannot find an IPv6 address, it tries to find an IPv4 address for loopback. The subroutine also clears the base structures that the SMUX code uses. The **smux_init** subroutine also sets the debug level that is used when it runs the SMUX subroutines.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
debug	Indicates the level of debug to be printed during SMUX subroutines.

Return Values

If the subroutine is successful, the socket descriptor is returned. Otherwise, the value of **NOTOK** is returned and the **smux_errno** global variable is set.

Error Codes

Possible values for the **smux_errno** global variable are:

Value	Description
congestion	Indicates memory allocation problems
youLoseBig	Signifies problem with SNMP library code
systemError	Indicates TCP connection failure.

These are defined in the **/usr/include/isode/snmp/smux.h** file.

Related information
List of Network Manager Programming References
SNMP Overview for Programmers

smux_register Subroutine

Purpose

Registers a section of the Management Information Base (MIB) tree with the Simple Network Management Protocol (SNMP) agent.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>
```

int smux_register (*subtree*, *priority*, *operation*)

```
OID subtree;
int priority;
int operation;
```

Description

The **smux_register** subroutine registers the section of the MIB tree for which the SMUX peer is responsible with the SNMP agent. Using the **smux_register** subroutine, the SMUX peer informs the SNMP agent of both the level of responsibility the SMUX peer has and the sections of the MIB tree for which it is responsible. The level of responsibility (priority) the SMUX peer sends determines which requests it can answer. Lower priority numbers correspond to higher priority.

If a tree is registered more than once, the SNMP agent sends requests to the registered SMUX peer with the highest priority. If the priority is set to -1, the SNMP agent attempts to give the SMUX peer the highest available priority. The *operation* parameter defines whether the MIB tree is added with readOnly or readWrite permissions, or if it should be deleted from the list of register trees. The SNMP agent returns an acknowledgment of the registration. The acknowledgment indicates the success of the registration and the actual priority received.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
subtree	Indicates an object identifier that contains the root of the MIB tree to be registered.
priority	Indicates the level of responsibility that the SMUX peer has on the MIB tree. The priority levels range from 0 to (2^31 - 2). The lower the priority number, the higher the priority. A priority of -1 tells the SNMP daemon to assign the highest priority currently available.
operation	Specifies the operation for the SNMP agent to apply to the MIB tree. Possible values are **delete**, **readOnly**, or **readWrite**. The **delete** operation removes the MIB tree from the SMUX peers in the eyes of the SNMP agent. The other two values specify the operations allowed by the SMUX peer on the MIB tree that is being registered with the SNMP agent.

Return Values

The values returned by this subroutine are **OK** on success and **NOTOK** on failure.

Error Codes

If the subroutine is unsuccessful, the **smux_errno** global variable is set to one of the following values:

Value	Description
parameterMissing	Indicates a parameter was null. When the parameter is fixed, the **smux_register** subroutine can be reissued.
invalidOperation	Indicates that the **smux_register** subroutine is trying to perform this operation before a **smux_init** operation has successfully completed. Start over with a new **smux_init** subroutine call.
congestion	Indicates a memory problem occurred. The TCP connection is closed. Start over with a new **smux_init** subroutine call.
youLoseBig	Indicates an SNMP code problem has occurred. The TCP connection is closed. Start over with a new **smux_init** subroutine call.

Related information

List of Network Manager Programming References

SNMP Overview for Programmers

smux_response Subroutine

Purpose

Sends a response to a Simple Network Management Protocol (SNMP) agent.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>
```

int smux_response (*event*)
struct type_SNMP_GetResponse__PDU ****event*;

Description

The **smux_response** subroutine sends a protocol data unit (PDU), also called an event, to the SNMP agent. The subroutine does not check whether the Management Information Base (MIB) tree is properly registered. The subroutine checks only to see whether a Transmission Control Protocol (TCP) connection to the SNMP agent exists and ensures that the *event* parameter is not null.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
event	Specifies a **type_SNMP_GetResponse__PDU** variable that contains the response PDU to send to the SNMP agent.

Return Values

If the subroutine is successful, **OK** is returned. Otherwise, **NOTOK** is returned.

Error Codes

If the subroutine is unsuccessful, the **smux_errno** global variable is set to one of the following values:

Value	Description
parameterMissing	Indicates the parameter was null. When the parameter is fixed, the subroutine can be reissued.
invalidOperation	Indicates the subroutine was attempted before the **smux_init** subroutine successfully completed. Start over with the **smux_init** subroutine.
youLoseBig	Indicates a SNMP code problem has occurred and the TCP connection is closed. Start over with the **smux_init** subroutine.

Related information

List of Network Manager Programming References

SNMP Overview for Programmers

smux_simple_open Subroutine

Purpose

Sends the open protocol data unit (PDU) to the Simple Network Management Protocol (SNMP) daemon.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>

int smux_simple_open (identity, description, commname, commlen)
OID  identity;
char * description;
char * commname;
int  commlen;
```

Description

Following the **smux_init** command, the **smux_simple_open** subroutine alerts the SNMP daemon that incoming messages are expected. Communication with the SNMP daemon is accomplished by sending an open PDU to the SNMP daemon. The **smux_simple_open** subroutine uses the *identity* object-identifier parameter to identify the SNMP multiplexing (SMUX) peer that is starting to communicate. The *description* parameter describes the SMUX peer. The *commname* and the *commlen* parameters supply the password portion of the open PDU. The *commname* parameter is the password used to authenticate the SMUX peer. The SNMP daemon finds the password in the **/etc/snmpd.conf** file. The SMUX peer can store the password in the **/etc/snmpd.peers** file. The *commlen* parameter specifies the length of the *commname* parameter value.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
identity	Specifies an object identifier that describes the SMUX peer.
description	Contains a string of characters that describes the SMUX peer. The *description* parameter value cannot be longer than 254 characters.
commname	Contains the password to be sent to the SNMP agent. Can be a null value.
commlen	Indicates the length of the community name (*commname* parameter) to be sent to the SNMP agent. The value for this parameter must be at least 0.

Return Values

The subroutine returns an integer value of **OK** on success or **NOTOK** on failure.

Error Codes

If the subroutine is unsuccessful, the **smux_errno** global variable is set one of the following values:

Value	Description
parameterMissing	Indicates that a parameter was null. The *commname* parameter can be null, but the *commlen* parameter value should be at least 0.

Value	Description
invalidOperation	Indicates that the **smux_init** subroutine did not complete successfully before the **smux_simple_open** subroutine was attempted. Correct the parameters and reissue the **smux_simple_open** subroutine.
inProgress	Indicates that the **smux_init** call has not completed the TCP connection. The **smux_simple_open** can be reissued.
systemError	Indicates the TCP connection was not completed. Do not reissue this subroutine without restarting the process with a **smux_init** subroutine call.
congestion	Indicates a lack of available memory space. Do not reissue this subroutine without restarting the process with a **smux_init** subroutine call.
youLoseBig	The SNMP code is having problems. Do not reissue this subroutine without restarting the process with a **smux_init** subroutine call.

Related information

List of Network Manager Programming References

SNMP Overview for Programmers

smux_trap Subroutine

Purpose

Sends SNMP multiplexing (SMUX) peer traps to the Simple Network Management Protocol (SNMP) agent.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>
```

int smux_trap (*generic*, *specific*, *bindings*)

```
int generic;
int specific;
struct type_SNMP_VarBindList *bindings;
```

Description

The **smux_trap** subroutine allows the SMUX peer to generate traps and send them to the SNMP agent. The subroutine sets the `generic` and `specific` fields in the trap packet to values specified by the parameters. The subroutine also allows the SMUX peer to send a list of variable bindings to the SNMP agent. The variable bindings are values associated with specific variables. If the trap is to return a set of variables, the variables are sent in the variable binding list.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
generic	Contains an integer specifying the generic trap type. The value must be one of the following: **0** Specifies a cold start. **1** Specifies a warm start. **2** Specifies a link down. **3** Specifies a link up. **4** Specifies an authentication failure. **5** Specifies an EGP neighbor loss. **6** Specifies an enterprise-specific trap type.
specific	Contains an integer that uniquely identifies the trap. The unique identity is typically assigned by the registration authority for the enterprise owning the SMUX peer.
bindings	Indicates the variable bindings to assign to the trap protocol data unit (PDU).

Return Values

The subroutine returns **NOTOK** on failure and **OK** on success.

Error Codes

If the subroutine is unsuccessful, the **smux_errno** global variable is set to one of the following values:

Value	Description
invalidOperation	Indicates the Transmission Control Protocol (TCP) connection was not completed.
congestion	Indicates memory is not available. The TCP connection was closed.
youLoseBig	Indicates an error occurred in the SNMP code. The TCP connection was closed.

Related information

List of Network Manager Programming References

SNMP Overview for Programmers

smux_wait Subroutine

Purpose

Waits for a message from the Simple Network Management Protocol (SNMP) agent.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/smux.h>
```

```
int smux_wait ( event,  isecs)
struct type_SMUX_PDUs **event;
int isecs;
```

Description

The **smux_wait** subroutine waits for a period of seconds, designated by the value of the *isecs* parameter, and returns the protocol data unit (PDU) received. The **smux_wait** subroutine waits on the socket descriptor that is initialized in a **smux_init** subroutine and maintained in the SMUX subroutines. The **smux_wait** subroutine waits up to *isecs* seconds. If the value of the *isecs* parameter is 0, the **smux_wait** subroutine returns only the first packet received. If the value of the *isecs* parameter is less than 0, the **smux_wait** subroutine waits indefinitely for the next message or returns a message already received. If no data is received, the **smux_wait** subroutine returns an error message of **NOTOK** and sets the **smux_errno** variable to the **inProgress** value. If the **smux_wait** subroutine is successful, it returns the first PDU waiting to be received. If a close PDU is received, the subroutine will automatically close the TCP connection and return **OK**.

This subroutine is part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
event	Points to a pointer of **type_SMUX_PDUs**. This holds the PDUs received by the **smux_wait** subroutine.
isecs	Specifies an integer value equal to the number of seconds to wait for a message.

Return Values

If the subroutine is successful, the value **OK** is returned. Otherwise, the return value is **NOTOK**.

Error Codes

If the subroutine is unsuccessful, the **smux_errno** global variable is set to one of the following values:

Value	Description
parameterMissing	Indicates that the *event* parameter value was null.
inProgress	Indicates that there was nothing for the subroutine to receive.
invalidOperation	Indicates that the **smux_init** subroutine was not called or failed to operate.
youLoseBig	Indicates an error occurred in the SNMP code. The TCP connection was closed.

Related information

List of Network Manager Programming References

SNMP Overview for Programmers

text2inst, name2inst, next2inst, or nextot2inst Subroutine

Purpose

Retrieves instances of variables from various forms of data.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/objects.h>
```

OI text2inst (*text*)
char ***text*;

OI name2inst (*oid*)
OID *oid*;

```
OI next2inst (oid)
OID oid;
```

OI nextot2inst (*oid*, *ot*)
OID *oid*;
OT *ot*;

Description

These subroutines return pointers to the actual objects in the database. When supplied with a way to identify the object, the subroutines return the corresponding object.

The **text2inst** subroutine takes a character string object identifier from the *text* parameter. The object's database is then examined for the specified object. If the specific object is not found, the **NULLOI** value is returned.

The **name2inst** subroutine uses an object identifier structure specified in the *oid* parameter to specify which object is desired. If the object cannot be found, a **NULLOI** value is returned.

The **next2inst** and **nextot2inst** subroutines find the next object in the database given an object identifier. The **next2inst** subroutine starts at the root of the tree, while the **nextot2inst** subroutine starts at the object given in the *ot* parameter. If another object cannot be found, the **NULLOI** value will be returned.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
text	Specifies the character string used to identify the object wanted in the **text2inst** subroutine.
oid	Specifies the object identifier structure used to identify the object wanted in the **name2inst**, **next2inst**, and **nextot2inst** subroutines.
ot	Specifies an object in the database used as a starting point for the **nextot2inst** subroutine.

Return Values

If the subroutine is successful, an **OI** value is returned. **OI** is a pointer to an object in the database. On a failure, a **NULLOI** value is returned.

Related reference
text2oid or text2obj Subroutine
Related information
List of Network Manager Programming References
SNMP Overview for Programmers

text2oid or text2obj Subroutine

Purpose

Converts a text string into some other value.

Library

SNMP Library (**libsnmp.a**)

Syntax

```
#include <isode/snmp/objects.h>
```

OID text2oid (*text*)
char ***text*;

```
OT text2obj (text)
char *text;
```

Description

The **text2oid** subroutine takes a character string and returns an object identifier. The string can be a name, a name**.**numbers, or dot notation. The returned object identifier is in memory-allocation storage and should be freed when the operation is completed with the **oid_free** subroutine.

The **text2obj** subroutine takes a character string and returns an object. The string needs to be the name of a specific object. The subroutine returns a pointer to the object.

These subroutines are part of the SNMP Application Programming Interface in the TCP/IP facility.

Parameters

Item	Description
text	Contains a text string used to specify the object identifier or object to be returned.

Return Values

On a successful execution, these subroutines return completed data structures. If a failure occurs, the **text2oid** subroutine returns a **NULLOID** value and the **text2obj** returns a **NULLOT** value.

Related reference
readobjects Subroutine
text2inst, name2inst, next2inst, or nextot2inst Subroutine
oid_cmp, oid_cpy, oid_free, sprintoid, str2oid, ode2oid, oid2ode, oid2ode_aux, prim2oid, or oid2prim Subroutine

Sockets

The operating system includes the Berkeley Software Distribution (BSD) interprocess communication (IPC) facility known as sockets. Sockets are communication channels that enable unrelated processes to exchange data locally and across networks. A single socket is one end point of a two-way communication channel. Socket subroutines enable interprocess and network interprocess communications (IPC).

_

AIX runtime services beginning with the character _.

_getlong Subroutine

Purpose

Retrieves long byte quantities.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

unsigned long _getlong (*MessagePtr***)**
u_char ****MessagePtr*;**

Description

The **_getlong** subroutine gets long quantities from the byte stream or arbitrary byte boundaries.

The **_getlong** subroutine is one of a set of subroutines that form the resolver, a set of functions that resolves domain names. Global information used by the resolver subroutines is kept in the **_res** data structure. The **/usr/include/resolv.h** file contains the **_res** structure definition.

All applications containing the **_getlong** subroutine must be compiled with the **_BSD** macro 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
MessagePtr	Specifies a pointer into the byte stream.

Return Values

The **_getlong** subroutine returns an unsigned long (32-bit) value.

Files

Item	Description
/etc/resolv.conf	Lists name server and domain names.

Related information
Sockets Overview
Understanding Domain Name Resolution

_getshort Subroutine

Purpose

Retrieves short byte quantities.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

unsigned short getshort ( MessagePtr)
u_char *MessagePtr;
```

Description

The **_getshort** subroutine gets quantities from the byte stream or arbitrary byte boundaries.

The **_getshort** subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the **_res** data structure. The **/usr/include/resolv.h** file contains the **_res** structure definition.

All applications containing the **_getshort** subroutine must be compiled with the **_BSD** macro 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
MessagePtr	Specifies a pointer into the byte stream.

Return Values

The **_getshort** subroutine returns an unsigned short (16-bit) value.

Files

Item	Description
/etc/resolv.conf	Defines name server and domain names.

Related information
Sockets Overview
Understanding Domain Name Resolution

_putlong Subroutine

Purpose

Places long byte quantities into the byte stream.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

void _putlong (*Long*, *MessagePtr*)
unsigned long *Long*;
**u_char ***MessagePtr*;

Description

The **_putlong** subroutine places long byte quantities into the byte stream or arbitrary byte boundaries.

The **_putlong** subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the **_res** data structure. The **/usr/include/resolv.h** file contains the **_res** structure definition.

All applications containing the **_putlong** subroutine must be compiled with the **_BSD** macro 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
Long	Represents a 32-bit integer.
MessagePtr	Represents a pointer into the byte stream.

Files

Item	Description
/etc/resolv.conf	Lists the name server and domain name.

Related information
Sockets Overview
Understanding Domain Name Resolution

_putshort Subroutine

Purpose

Places short byte quantities into the byte stream.

Library

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

Syntax

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

```
#include <arpa/nameser.h>
#include <resolv.h>
```

void _putshort (*Short*, *MessagePtr*)
unsigned short *Short*;
u_char **MessagePtr*;

Description

The **_putshort** subroutine puts short byte quantities into the byte stream or arbitrary byte boundaries.

The **_putshort** subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the **_res** data structure. The **/usr/include/resolv.h** file contains the **_res** structure definition.

All applications containing the **_putshort** subroutine must be compiled with the **_BSD** macro 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
Short	Represents a 16-bit integer.
MessagePtr	Represents a pointer into the byte stream.

Files

Item	Description
/etc/resolv.conf	Lists the name server and domain name.

Related information
Sockets Overview
Understanding Domain Name Resolution

a

AIX runtime services beginning with the letter *a*.

accept Subroutine

Purpose

Accepts a connection on a socket to create a new socket.

Library

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

Syntax

#include <sys/socket.h>

int accept (*Socket*, *Address*, *AddressLength*)
int *Socket*;
struct sockaddr **Address*;
socklen_t **AddressLength*;

Description

The **accept** subroutine extracts the first connection on the queue of pending connections, creates a new socket with the same properties as the specified socket, and allocates a new file descriptor for that socket.

If the **listen** queue is empty of connection requests, the **accept** subroutine:

- Blocks a calling socket of the blocking type until a connection is present.
- Returns an **EWOULDBLOCK** error code for sockets marked nonblocking.

The accepted socket cannot accept more connections. The original socket remains open and can accept more connections.

The **accept** subroutine is used with **SOCK_STREAM** and **SOCK_CONN_DGRAM** socket types.

For **SOCK_CONN_DGRAM** socket type and **ATM** protocol, a socket is not ready to transmit/receive data until **SO_ATM_ACCEPT** socket option is called. This allows notification of an incoming connection to the application, followed by modification of appropriate parameters and then indicate that a connection can become fully operational.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies a socket created with the **socket** subroutine that is bound to an address with the **bind** subroutine and has issued a successful call to the **listen** subroutine.
Address	Specifies a result parameter that is filled in with the address of the connecting entity as known to the communications layer. The exact format of the *Address* parameter is determined by the domain in which the communication occurs.
AddressLength	Specifies a parameter that initially contains the amount of space pointed to by the *Address* parameter. Upon return, the parameter contains the actual length (in bytes) of the address returned. The **accept** subroutine is used with **SOCK_STREAM** socket types.

Return Values

Upon successful completion, the **accept** subroutine returns the nonnegative socket descriptor of the accepted socket.

If the **accept** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **ernno** global variable.

Error Codes

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

Item	Description
EBADF	The *Socket* parameter is not valid.
ECONNRESET	The connection has been reset by the partner.
EINTR	The **accept** function was interrupted by a signal that was caught before a valid connection arrived.

Item	Description
EINVAL	The socket referenced by *s* is not currently a listen socket or has been shutdown with **shutdown**. A **listen** must be done before an **accept** is allowed.
EMFILE	The system limit for open file descriptors per process has already been reached (**OPEN_MAX**).
ENFILE	The maximum number of files allowed are currently open.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The referenced socket is not of type **SOCK_STREAM**.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EWOULDBLOCK	The socket is marked as nonblocking, and no connections are present to be accepted.
ENETDOWN	The network with which the socket is associated is down.
ENOTCONN	The socket is not in the connected state.
ECONNABORTED	The client aborted the connection.

Examples

As illustrated in this program fragment, once a socket is marked as listening, a server process can accept a connection:

```
struct sockaddr_in from;
.
.
.
fromlen = sizeof(from);
newsock = accept(socket, (struct sockaddr*)&from, &fromlen);
```

Related reference
listen Subroutine
socket Subroutine
Related information
Accepting Stream Connections Example Program
Binding Names to Sockets

arpresolve_common Subroutine

Purpose

Reads or creates new arp entries so that hardware addresses can be resolved.

Syntax

```
int  arpresolve_common ( ac, m, arpwhohas, dst, hwaddr, szhwaddr, extra, if_dependent)

        register struct arpcom *ac;
        struct mbuf *m;
        int (*arpwhohas)(register struct arpcom *ac,
                         struct in_addr *addr, int skipbestif, void *extra),
        struct sockaddr_in *dst;
        u_char * hwaddr;
        int szhwaddr;
        void *extra;
        union if_dependent *if_dependent;
```

Description

The **arpresolve_common** subroutine reads or creates new arp entries so that hardware addresses can be resolved. It is called by **arpresolve** from the IF layer of the interface. If the arp entry is complete, then **arpresolve_common** returns the address pointed to by *hwaddr* and the data pointed to by *if_dependent* if *if_dependent* is true. If the arp entry is not complete, then this subroutine adds the memory buffer pointed to by *mbuf* to **at_hold**. **at_hold** holds one or more packets that are waiting for the arp entry to complete so they can be transmitted.

If an arp entry does not exist, **arpresolve_common** creates a new entry by calling **arptnew** and then adds the memory buffer pointed to by *mbuf* to **at_hold**. This subroutine calls **arpwhohas** when it creates a new arp entry or when the timer for the incomplete arp entry (with the IP address that is pointed to by *dst*) has expired.

Parameters

Item	Description
ac	Points to the **arpcom** structure.
m	Points to the memory buffer (**mbuf**), which will be added to the list awaiting completion of the arp table entry.
arpwhohas	Points to the **arpwhohas** subroutine.
addr	Points to the **in_addr** structure's address.
extra	A void pointer that can be used in the future so that IF layers can pass extra structures to **arpwhohas**.
dst	Points to the **sockaddr_in** structure. This structure has the destination IP address.
hwaddr	Points to the buffer. This buffer contains the hardware address if it finds a completed entry.
szhwaddr	Size of the buffer pointed to by *hwaddr*.
if_dependent	Pointer to the **if_dependent** structure. **arpresolve_common** uses this to pass the **if_dependent** data, which is part of the **arptab** entry, to the calling function.

Return Values

Item	Description
ARP_MBUF	The arp entry is not complete.
ARP_HWADDR	The *hwaddr* buffer is filled with the hardware address.
ARP_FLG_NOARP	The arp entry does not exist, and the **IFF_NOARP** flag is set only if the value of **if_type** is **IFT_ETHER**.

arpupdate Subroutine

Purpose

Updates arp entries for a given IP address.

Syntax

```
int arpupdate (ac, m, hp, action, prm)
      register struct arpcom *ac;
```

```
struct mbuf *m;
caddr_t hp;
int action;
struct arpupdate_parm *prm;
```

Description

The **arpupdate** subroutine updates arp entries for a given IP address. It is called by **arpinput** from the IF layer of the interface. This subroutine searches the arp table for an entry that matches the IP address. It then updates the arp entry for the given IP address. The **arpupdate** subroutine also performs reverse arp lookups.

The **arpupdate** subroutine enters a new address in **arptab**, pushing out the oldest entry from the bucket if there is no room. This subroutine always succeeds because no bucket can be completely filled with permanent entries (except when **arpioctl** tests whether another permanent entry can fit).

Depending on the action specified, the prm IP addresses **isaddr**, **itaddr**, and **myaddr** are used by the **arpupdate** subroutine.

Parameters

Item	Description
ac	Points to the **arpcom** structure.
m	Points to the memory buffer (**mbuf**), that contains the arp response packet received by the interface.
hp	Points to the buffer that is passed by the interrupt handler.
action	Returns a value that indicates which action is taken: **LOOK** Looks for the **isaddr** IP address in the arp table and returns the hardware address and **if_dependent** structure. **LKPUB** Looks for the **isaddr** IP address in the arp table and returns the hardware address and **if_dependent** structure only if the **ATF_PUBL** is set. **UPDT** Updates the arp entry for an IP address (**isaddr**). If no arp entry is there, creates a new one and updates the **if_dependent** structure using the **ptr** function passed in the **prm** structure. **REVARP** Reverses the arp request. *hwaddr* contains the hardware address, *szhwaddr* indicates its size, and *saddr* returns the IP address if an entry is found.
prm	Points to the **arpudpate_parm** structure. The values are: **LOOK or LKPUB** **itaddr** and **myaddr** are ignored. **isaddr** is used for arp table lookup. **UPDTE** **isaddr** points to the sender protocol address. **itaddr** points to the target protocol address. **myaddr** points to the protocol address of the interface that received the packet.

Return Values

Item	Description
ARP_OK	Lookup or update was successful.

Item	Description
ARP_FAIL	Lookup or update failed.
ARP_NEWF	New arp entry could not be created.

b

AIX runtime services beginning with the letter *b*.

bind Subroutine

Purpose

Binds a name to a socket.

Library

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

Syntax

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

```
int bind ( Socket,  Name,  NameLength)
int Socket;
const struct sockaddr *Name;
socklen_t NameLength;
```

Description

The **bind** subroutine assigns a *Name* parameter to an unnamed socket. Sockets created by the **socket** subroutine are unnamed; they are identified only by their address family. Subroutines that connect sockets either assign names or use unnamed sockets.

For a UNIX domain socket, a **connect** call only succeeds if the process that calls **connect** has read and write permissions on the socket file created by the **bind** call. Permissions are determined by the **umask** value of the process that created the file.

An application program can retrieve the assigned socket name with the **getsockname** subroutine.

The socket applications can be compiled with **COMPAT_43** defined. This makes the **sockaddr** structure BSD 4.3 compatible. For more details refer to the **socket.h** file.

Binding a name in the UNIX domain creates a socket in the file system that must be deleted by the caller when it is no longer needed.

Note: When you enable IPv6 for an application, IPv4 addresses are also supported. You can use an AF_INET6 socket to send and receive both IPv4 and IPv6 packets because AF_INET6 sockets are capable of handling communication with both IPv4 and IPv6 hosts. However, you must convert the address format of the IPv4 addresses that were previously passed to the socket calls to the IPv4-mapped IPv6 address format. For example, you must convert 10.1.1.1 in the *sockaddr_in* structure to ::ffff:10.1.1.1 in the *sockaddr_in6* structure.

Parameters

Item	Description
Socket	Specifies the socket descriptor (an integer) of the socket to be bound.

Item	Description
Name	Points to an address structure that specifies the address to which the socket should be bound. The **/usr/include/sys/socket.h** file defines the **sockaddr** address structure. The **sockaddr** structure contains an identifier specific to the address format and protocol provided in the **socket** subroutine.
NameLength	Specifies the length of the socket address structure.

Return Values

Upon successful completion, the **bind** subroutine returns a value of 0.

If the **bind** subroutine is unsuccessful, the subroutine handler performs the following actions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see "Error Notification Object Class" in *Communications Programming Concepts*.

Error Codes

The **bind** subroutine is unsuccessful if any of the following errors occurs:

Value	Description
EACCES	The requested address is protected, and the current user does not have permission to access it.
EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The specified address is not a valid address for the address family of the specified socket.
EAGAIN	The transient ports are already in use and are not available.
EBADF	The *Socket* parameter is not valid.
EDESTADDRREQ	The *address* argument is a null pointer.
EFAULT	The *Address* parameter is not in a writable part of the *UserAddress* space.
EINVAL	The socket is already bound to an address.
ENOBUF	Insufficient buffer space available.
ENODEV	The specified device does not exist.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The socket referenced by *Socket* parameter does not support address binding.

Examples

The following program fragment illustrates the use of the **bind** subroutine to bind the name "/tmp/zan/" to a UNIX domain socket.

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

```
.
.
.
struct sockaddr_un addr;
.
.
```

```
.
strcpy(addr.sun_path, "/tmp/zan/");
addr.sun_len = strlen(addr.sun_path);
addr.sun_family = AF_UNIX;
bind(s,(struct sockaddr*)&addr, SUN_LEN(&addr));
```

Related reference
connect Subroutine
socket Subroutine

bind2addrsel Subroutine

Purpose

Binds a socket to an address according to address selection preferences.

Library

Library (**libc.a**)

Syntax

```
#include <netinet/in.h>
int bind2addrsel(int socket,const struct sockaddr *dstaddr, socklen_t dstaddrlen)
```

Description

When establishing a communication with a distant address, AIX uses a address selection algorithm to define what local address will be used to communicate with a distant address. This algorithm uses a set of ordered rules (RFC 3484) to choose this local address. Some of these rules use the type of address for this selection. By default, public addresses are preferred over temporary addresses; CGA addresses are preferred over non CGA addresses; home addresses are preferred over care-of addresses. An application may prefer the use other preference choices (for example use a temporary address rather that a public address) for the rules using the type of address. If these rules are applied, these preferences will be used. The application can express these preferences using a **setsockopt** call with the IPV6_ADDR_PREFERENCES option and a combination of the following flags:

- IPV6_PREFER_SRC_HOME: prefer addresses reachable from a Home source address
- IPV6_PREFER_SRC_COA: prefer addresses reachable from a Care-of source address
- IPV6_PREFER_SRC_TMP: prefer addresses reachable from a temporary address
- IPV6_PREFER_SRC_PUBLIC: the prefer addresses reachable from a public source address
- IPV6_PREFER_SRC_CGA: the prefer addresses reachable from a Cryptographically Generated Address (CGA) source address
- IPV6_PREFER_SRC_NONCGA: the prefer addresses reachable from a non-CGA source address

The application will then call **bind2addrsel. bind2addrsel** binds a socket to a local address selected to communicate with the given destination address according to the address selection preferences.

Parameters

Item	Description
socket	Specifies the unique socket name
dstaddr	Points to a sockaddr structure containing the destination address. The sin6_family field of this sockaddr structure must be set to AF_INET6.
dstaddrlen	Specifies the size of the sockaddr structure pointed by dstaddr.

Return Values

Upon successful completion, the subroutine returns 0

If unsuccessful, the subroutine returns -1 and errno is set accordingly:

c

AIX runtime services beginning with the letter *c*.

connect Subroutine

Purpose

Connects two sockets.

Library

Standard C Library (**libc.a**

Syntax

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

```
int connect ( Socket, Name,  NameLength)
int Socket;
const struct sockaddr *Name;
socklen_t NameLength;
```

Description

The **connect** subroutine requests a connection between two sockets. The kernel sets up the communication link between the sockets; both sockets must use the same address format and protocol.

If a **connect** subroutine is issued on an unbound socket or a partially bound socket (a socket that is assigned a port number but no IP address), the system automatically binds the socket. The `connect` subroutine can be used to connect a socket to itself. This can be done, for example, by binding a socket to a local port (using `bind`) and then connecting it to the same port with a local IP address (using `connect`).

The **connect** subroutine performs a different action for each of the following two types of initiating sockets:

- If the initiating socket is **SOCK_DGRAM**, the **connect** subroutine establishes the peer address. The peer address identifies the socket where all datagrams are sent on subsequent **send** subroutines. No connections are made by this **connect** subroutine. If the UDP socket is receiving datagrams when the **connect** subroutine is called, the subroutine will change the IP address, preventing the socket from receiving datagram packets based on the previous address.
- If the initiating socket is **SOCK_STREAM** or **SOCK_CONN_DGRAM**, the **connect** subroutine attempts to make a connection to the socket specified by the *Name* parameter. Each communication space interprets the *Name* parameter differently. For **SOCK_CONN_DGRAM** socket type and ATM protocol, some of the ATM parameters may have been modified by the remote station, applications may query new values of ATM parameters using the appropriate socket options.
- In the case of a UNIX domain socket, a **connect** call only succeeds if the process that calls **connect** has read and write permissions on the socket file created by the **bind** call. Permissions are determined by the **umask<** value of the process that created the file.

Implementation Specifics

Parameters

Item	Description
Socket	Specifies the unique name of the socket.
Name	Specifies the address of target socket that will form the other end of the communication line
NameLength	Specifies the length of the address structure.

Return Values

Upon successful completion, the **connect** subroutine returns a value of 0.

If the **connect** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **connect** subroutine is unsuccessful if any of the following errors occurs:

Value	Description
EADDRINUSE	The specified address is already in use. This error will also occur if the **SO_REUSEADDR** socket option was set and the local address (whether specified or selected by the system) is already in use.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EALREADY	The socket is specified with **O_NONBLOCK** or **O_NDLAY**, and a previous connecttion attempt has not yet completed.
EINTR	The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection will be established asynchronously.
EACCES	Search permission is denied on a component of the path prefix or write access to the named socket is denied.
ENOBUFS	The system ran out of memory for an internal data structure.
EOPNOTSUPP	The socket referenced by *Socket* parameter does not support **connect**.
EWOULDBLOCK	The range allocated for TCP/UDP ephemeral ports has been exhausted.
EBADF	The *Socket* parameter is not valid.
ECONNREFUSED	The attempt to connect was rejected.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINPROGRESS	The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.
EINVAL	The specified path name contains a character with the high-order bit set.
EISCONN	The socket is already connected.
ENETDOWN	The specified physical network is down.
ENETUNREACH	No route to the network or host is present.
ENOSPC	There is no space left on a device or system table.

Value	Description
ENOTCONN	The socket could not be connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ETIMEDOUT	The establishment of a connection timed out before a connection was made.
EPROTOTYPE	The specified address has a different type from the socket that is bound to the specified peer address.
ELOOP	Too many symbolic links were encountered in translating the path name in address.
ENOENT	A component of the path name does not name an existing file or the path name is an empty string.
ENOTDIR	A component of the path prefix of the path name in address is not a directory.

Examples

The following program fragment illustrates the use of the **connect** subroutine by a client to initiate a connection to a server's socket.

```
struct sockaddr_un server;
.
.
.
connect(s,(struct sockaddr*)&server, sun_len(&server));
```

Related reference
bind Subroutine
/etc/socks5c.conf File

CreateIoCompletionPort Subroutine

Purpose

Creates an I/O completion port with no associated file descriptor or associates an opened socket or file with an existing or newly created I/O completion port.

Syntax

```
#include <iocp.h>
int CreateIoCompletionPort (FileDescriptor, CompletionPort, CompletionKey, ConcurrentThreads)
HANDLE FileDescriptor, CompletionPort;
DWORD CompletionKey, ConcurrentThreads;
```

Description

The **CreateIoCompletionPort** subroutine creates an I/O completion port or associates an open file descriptor with an existing or newly created I/O completion port. When creating a new I/O completion port, the *CompletionPort* parameter is set to NULL, the *FileDescriptor* parameter is set to INVALID_HANDLE_VALUE (-1), and the *CompletionKey* parameter is ignored.

The **CreateIoCompletionPort** subroutine returns a descriptor (an integer) to the I/O completion port created or modified.

The **CreateIoCompletionPort** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note: This subroutine only works with file descriptors of sockets, or regular files for use with the Asynchronous I/O (AIO) subsystem. It does not work with file descriptors of other types.

Parameters

Item	Description
FileDescriptor	Specifies a valid file descriptor obtained from a call to the **socket** or **accept** subroutines.
CompletionPort	Specifies a valid I/O completion port descriptor. Specifying a *CompletionPort* parameter value of NULL causes the **CreateIoCompletionPort** subroutine to create a new I/O completion port.
CompletionKey	Specifies an integer to serve as the identifier for completion packets generated from a particular file-completion port set.
ConcurrentThreads	This parameter is not implemented and is present only for compatibility purposes.

Return Values

Upon successful completion, the **CreateIoCompletionPort** subroutine returns an integer (the I/O completion port descriptor).

If the **CreateIoCompletionPort** is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of NULL to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

Error Codes

The **CreateIoCompletionPort** subroutine is unsuccessful if either of the following errors occur:

Item	Description
EBADF	The I/O completion port descriptor is invalid.
EINVAL	The file descriptor is invalid.
EALREADY	The file descriptor is already associated.

Examples

The following program fragment illustrates the use of the **CreateIoCompletionPort** subroutine to create a new I/O completion port with no associated file descriptor:

```
c = CreateIoCompletionPort (INVALID_HANDLE_VALUE, NULL, 0, 0);
```

The following program fragment illustrates the use of the **CreateIoCompletionPort** subroutine to associate file descriptor 34 (which has a newly created I/O completion port) with completion key 25:

```
c = CreateIoCompletionPort (34, NULL, 25, 0);
```

The following program fragment illustrates the use of the **CreateIoCompletionPort** subroutine to associate file descriptor 54 (which has an existing I/O completion port) with completion key 15:

```
c = CreateIoCompletionPort (54, 12, 15, 0);
```

Related information

Error Notification Object Class

d

AIX runtime services beginning with the letter *d*.

dn_comp Subroutine

Purpose

Compresses a domain name.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

int dn_comp (*ExpDomNam***, *CompDomNam*, *Length*, *DomNamPtr*, *LastDomNamPtr*)**
u_char * *ExpDomNam*, * *CompDomNam*;
int *Length*;
u_char ** *DomNamPtr*, ** *LastDomNamPtr*;

Description

The **dn_comp** subroutine compresses a domain name to conserve space. When compressing names, the client process must keep a record of suffixes that have appeared previously. The **dn_comp** subroutine compresses a full domain name by comparing suffixes to a list of previously used suffixes and removing the longest possible suffix.

The **dn_comp** subroutine compresses the domain name pointed to by the *ExpDomNam* parameter and stores it in the area pointed to by the *CompDomNam* parameter. The **dn_comp** subroutine inserts labels into the message as the name is compressed. The **dn_comp** subroutine also maintains a list of pointers to the message labels and updates the list of label pointers.

- If the value of the *DomNamPtr* parameter is null, the **dn_comp** subroutine does not compress any names. The **dn_comp** subroutine translates a domain name from ASCII to internal format without removing suffixes (compressing). Otherwise, the *DomNamPtr* parameter is the address of pointers to previously compressed suffixes.
- If the *LastDomNamPtr* parameter is null, the **dn_comp** subroutine does not update the list of label pointers.

The **dn_comp** subroutine is one of a set of subroutines that form the resolver. The resolver is a set of functions that perform a translation between domain names and network addresses. Global information used by the resolver subroutines resides in the **_res** data structure. The **/usr/include/resolv.h** file contains the **_res** data structure definition.

All applications containing the **dn_comp** subroutine must be compiled with the **_BSD** macro 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
ExpDomNam	Specifies the address of an expanded domain name.

Item	Description
CompDomNam	Points to an array containing the compressed domain name.
Length	Specifies the size of the array pointed to by the *CompDomNam* parameter.
DomNamPtr	Specifies a list of pointers to previously compressed names in the current message.
LastDomNamPtr	Points to the end of the array specified to by the *CompDomNam* parameter.

Return Values

Upon successful completion, the **dn_comp** subroutine returns the size of the compressed domain name.

If unsuccessful, the **dn_comp** subroutine returns a value of -1 to the calling program.

Files

Item	Description
/usr/include/resolv.h	Contains global information used by the resolver subroutines.

Related reference
dn_expand Subroutine
Related information
TCP/IP name resolution
Sockets Overview
Understanding Domain Name Resolution

dn_expand Subroutine

Purpose

Expands a compressed domain name.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

```
int dn_expand (MessagePtr, EndofMesOrig, CompDomNam, ExpandDomNam, Length)
u_char * MessagePtr, * EndOfMesOrig;
u_char * CompDomNam, * ExpandDomNam;
int  Length;
```

Description

The **dn_expand** subroutine expands a compressed domain name to a full domain name, converting the expanded names to all uppercase letters. A client process compresses domain names to conserve space. Compression consists of removing the longest possible previously occurring suffixes. The **dn_expand** subroutine restores a domain name compressed by the **dn_comp** subroutine to its full size.

The **dn_expand** subroutine is one of a set of subroutines that form the resolver. The resolver is a set of functions that perform a translation between domain names and network addresses. Global information used by the resolver subroutines resides in the **_res** data structure. The **/usr/include/resolv.h** file contains the **_res** data structure definition.

All applications containing the **dn_expand** subroutine must be compiled with the **_BSD** macro 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
MessagePtr	Specifies a pointer to the beginning of a message.
EndOfMesOrig	Points to the end of the original message that contains the compressed domain name.
CompDomNam	Specifies a pointer to a compressed domain name.
ExpandDomNam	Specifies a pointer to a buffer that holds the resulting expanded domain name.
Length	Specifies the size of the buffer pointed to by the *ExpandDomNam* parameter.

Return Values

Upon successful completion, the **dn_expand** subroutine returns the size of the expanded domain name.

If unsuccessful, the **dn_expand** subroutine returns a value of -1 to the calling program.

Files

Item	Description
/etc/resolv.conf	Defines name server and domain name constants, structures, and values.

Related reference
dn_comp Subroutine
Related information
exit subroutine
TCP/IP name resolution

e

AIX runtime services beginning with the letter *e*.

eaccept Subroutine

Purpose

Accepts a connection on a socket to create a new socket. The **eaccept** subroutine is similar to the **accept** subroutine with the addition of the **sec_labels_t** structure. The **sec_labels_t** structure reads the Sensitivity Level (SL) that is received on the incoming connection for Trusted AIX enabled systems.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/mac.h>
```

```
int eaccept ( Socket,  Address,  AddressLength, Label)
int Socket;
struct sockaddr *Address;
socklen_t *AddressLength;
sec_labels_t *Label;
```

Description

The **eaccept** subroutine extracts the first connection in the queue of pending connections, creates a new socket with the same properties as the specified socket, and allocates a new file descriptor for that socket.

If there are no connection requests in the **listen** queue, the **eaccept** subroutine performs the following actions:

- Blocks a calling socket of the **blocking** type until a connection is present.
- Returns an **EWOULDBLOCK** error code for sockets marked nonblocking.

The accepted socket cannot accept more connections, but the original socket remains open and can accept more connections.

The **eaccept** subroutine is used with only the SOCK_STREAM socket type. If a valid *Label* parameter is specified, the SL from the incoming connection is returned to the application.

Parameters

Item	Description
Socket	Specifies a socket created with the **socket** subroutine that is bound to an address with the **bind** or **ebind** subroutine and has issued a successful call to the **listen** subroutine.
Address	Specifies a result parameter that contains the address of the connecting entity as known to the communications layer. The exact format of the *Address* parameter is determined by the domain in which the communication occurs.
AddressLength	Specifies a parameter that initially contains the amount of space pointed to by the *Address* parameter. Upon return, the parameter contains the actual length (in bytes) of the address returned. The **eaccept** subroutine is used with **SOCK_STREAM** socket types.
Label	Specifies a result parameter that contains the SL received on the incoming connection.

Return Values

Item	Description
Successful	a non-negative socket descriptor of the accepted socket
Unsuccessful	-1

Error Codes

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

Item	Description
EBADF	The *Socket* parameter is not valid.
EINTR	The **eaccept** function was interrupted by a signal that was caught before a valid connection arrived.
EINVAL	The socket referenced by *s* is not currently a listen socket or has been shutdown with **shutdown**. A **listen** must be done before an **eaccept** is allowed.
EMFILE	The number of open file descriptors per process exceeds the system limit (**OPEN_MAX**).
ENFILE	The number of open files exceeds the allowed maximum value.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The referenced socket is not of type **SOCK_STREAM**.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EWOULDBLOCK	The socket is marked as nonblocking, and no connections are present to be accepted.
ENETDOWN	The network that the socket is associated with is down.
ENOTCONN	The socket is not in the connected state.
ECONNABORTED	The client aborted the connection.
EPERM	The MLS MAC check failed.

ebind Subroutine

Purpose

Binds a name to a socket. Also binds a socket to the specific Sensitivity Level (SL) that is passed as a parameter.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/mac.h>
```

```
int ebind ( Socket, Name, NameLength, Label)
int Socket;
const struct sockaddr *Name;
socklen_t NameLength;
sec_labels_t *Label;
```

Description

The **ebind** subroutine assigns a *Name* parameter to an unnamed socket. Sockets created by the **socket** subroutine are unnamed; they are identified only by their address family. Subroutines that connect sockets either assign names or use unnamed sockets.

When a NULL pointer is passed to the *Label* parameter, then a normal multi-level port is created. However, when a valid label is passed to the *Label* parameter, a port at the specified Sensitivity Level (SL) is created. This means that only those incoming connections at the specified SL are able to connect. This also means

that multiple sockets can be bound to the same port at different SLs. It is possible to create a multi-level port as well as several specific-level ports. If none of the specific SLs matches the incoming packet, then the packet port is a default multi-level port.

Parameters

Item	Description
Socket	Specifies the socket descriptor of the socket to be bound. The socket descriptor is an integer,
Name	Points to an address structure that specifies the address to which the socket should be bound. The **/usr/include/sys/socket.h** file defines the **sockaddr** address structure. The **sockaddr** structure contains an identifier specific to the address format and protocol provided in the **socket** subroutine.
NameLength	Specifies the length of the socket address structure.
Label	Specifies the Sensitivity Label associated with the socket.

Return Values

Item	Description
Successful	0
Unsuccessful	-1

Error Codes

The **ebind** subroutine is unsuccessful if any of the following errors occurs:

Value	Description
EACCES	The requested address is protected, and the current user does not have permission to access it.
EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The specified address is not a valid address for the address family of the specified socket.
EBADF	The *Socket* parameter is not valid.
EDESTADDRREQ	The *address* argument is a null pointer.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINVAL	The socket is already bound to an address.
ENOBUF	Insufficient buffer space available.
ENODEV	The specified device does not exist.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The socket referenced by the *Socket* parameter does not support address binding.

econnect Subroutine

Purpose

Connects two sockets. The **econnect** subroutine is similar to the **connect** subroutine with the addition of the **sec_labels_t** pointer. The **sec_labels_t** pointer indicates the Sensitivity Level (SL) of the outgoing connection request.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/mac.h>
```

```
int econnect ( Socket, Name, NameLength, Label)
int Socket;
const struct sockaddr *Name;
socklen_t NameLength;
sec_labels_t *Label;
```

Description

The **econnect** subroutine requests a connection between two sockets, similar to the **connect** subroutine. The kernel sets up the communication link between the sockets; both sockets must use the same address format and protocol.

The SL specified by the *Label* parameter is the SL of the outgoing request. The requested SL must be dominated by the current clearance or must have appropriate privileges to clear the MAC check.

Parameters

Item	Description
Socket	Specifies the unique name of the socket.
Name	Specifies the address of the target socket that will form the other end of the communication line.
NameLength	Specifies the length of the address structure.
Label	Specifies the SL of the outgoing connection request.

Return Values

Item	Description
Successful	0
Unsuccessful	-1

Error Codes

The **econnect** subroutine is unsuccessful if any of the following errors occurs:

Value	Description
EADDRINUSE	The specified address is already in use.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.

Value	Description
EALREADY	The socket is specified with **O_NONBLOCK** or **O_NDLAY**, and a previous connection attempt has not yet completed.
EINTR	The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection will be established asynchronously.
EACCES	Search permission was denied on a component of the path prefix or write access to the named socket was denied.
ENOBUFS	The system has run out of memory for an internal data structure.
EOPNOTSUPP	The socket referenced by the *Socket* parameter does not support the **econnect** subroutine.
EWOULDBLOCK	The range allocated for TCP/UDP ephemeral ports has been exhausted.
EBADF	The *Socket* parameter is not valid.
ECONNREFUSED	The attempt to connect was rejected.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINPROGRESS	The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.
EINVAL	The specified path name contains a character with the high-order bit set.
EISCONN	The socket is already connected.
ENETDOWN	The specified physical network is down.
ENETUNREACH	No route to the network or host is present.
ENOSPC	There is no space left on a device or system table.
ENOTCONN	The socket could not be connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ETIMEDOUT	The establishment of a connection times out before a connection is made.
EPERM	The Trusted AIX MAC check failed.

endhostent Subroutine

Purpose

Closes the **/etc/hosts** file.

Library

```
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>
endhostent ()
```

Description

When using the **endhostent** subroutine in DNS/BIND name service resolution, **endhostent** closes the TCP connection which the **sethostent** subroutine set up.

When using the **endhostent** subroutine in NIS name resolution or to search the **/etc/hosts** file, **endhostent** closes the **/etc/hosts** file.

Note: If a previous **sethostent** subroutine is performed and the *StayOpen* parameter does not equal 0, the **endhostent** subroutine closes the **/etc/hosts** file. Run a second **sethostent** subroutine with the *StayOpen* value equal to 0 in order for a following **endhostent** subroutine to succeed. Otherwise, the **/etc/hosts** file closes on an **exit** subroutine call .

Files

Item	Description
/etc/hosts	Contains the host name database.
/etc/netsvc.conf	Contains the name service ordering.
/usr/include/netdb.h	Contains the network database structure.

Related reference
sethostent Subroutine
Related information
exit subroutine
Sockets Overview

endhostent_r Subroutine

Purpose

Closes the **/etc/hosts** file.

Library

```
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>

void endhostent_r (struct hostent_data *ht_data);
```

Description

When using the **endhostent_r** subroutine in DNS/BIND name service resolution, **endhostent_r** closes the TCP connection which the **sethostent_r** subroutine set up.

When using the **endhostent_r** subroutine in NIS name resolution or to search the **/etc/hosts** file, **endhostent_r** closes the **/etc/hosts** file.

Note: If a previous **sethostent_r** subroutine is performed and the *StayOpen* parameter does not equal 0, then the **endhostent_r** subroutine closes the **/etc/hosts** file. Run a second **sethostent_r** subroutine with the *StayOpen* value equal to 0 in order for a following **endhostent_r** subroutine to succeed. Otherwise, the **/etc/hosts** file closes on an **exit** subroutine call .

Parameters

Item	Description
ht_data	Points to the **hostent_data** structure

Files

Item	Description
/etc/hosts	Contains the host name database.
/etc/netsvc.conf	Contains the name service ordering.
/usr/include/netdb.h	Contains the network database structure.

endnetent Subroutine

Purpose

Closes the **/etc/networks** file.

Library

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

Syntax

```
#include <netdb.h>
```

```
void endnetent ( )
```

Description

The **endnetent** subroutine closes the **/etc/networks** file. Calls made to the **getnetent**, **getnetbyaddr**, or **getnetbyname** subroutine open the **/etc/networks** file.

All applications containing the **endnetent** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Return Values

If a previous **setnetent** subroutine has been performed and the *StayOpen* parameter does not equal 0, then the **endnetent** subroutine will not close the **/etc/networks** file. Also, the **setnetent** subroutine does not indicate that it closed the file. A second **setnetent** subroutine has to be issued with the *StayOpen* parameter equal to 0 in order for a following **endnetent** subroutine to succeed. If this is not done, the **/etc/networks** file must be closed with the **exit** subroutine.

Examples

To close the **/etc/networks** file, type:

```
endnetent();
```

Files

Item	Description
/etc/networks	Contains official network names.

Related reference
setnetent Subroutine
Related information
exit subroutine

endnetent_r Subroutine

Purpose

Closes the **/etc/networks** file.

Library

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

Syntax

```
#include <netdb.h>

void endnetent_r (net_data)
struct netent_data *net_data;
```

Description

The **endnetent_r** subroutine closes the **/etc/networks** file. Calls made to the **getnetent_r**, **getnetbyaddr_r**, or **getnetbyname_r** subroutine open the **/etc/networks** file.

Parameters

Item	Description
net_data	Points to the **netent_data** structure.

Files

Item	Description
/etc/networks	Contains official network names.

endnetgrent_r Subroutine

Purpose

Handles the group network entries.

Library

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

Syntax

```
#include <netdb.h>
 void endnetgrent_r (void **ptr)
```

Description

The **setnetgrent_r** subroutine establishes the network group from which the **getnetgrent_r** subroutine will obtain members, and also restarts calls to the **getnetgrent_r** subroutine from the beginnning of the list. If the previous **setnetgrent_r** call was to a different network group, an **endnetgrent_r** call is implied.

The **endnetgrent_r** subroutine frees the space allocated during the **getnetgrent_r** calls.

Parameters

Item	Description
ptr	Keeps the function threadsafe.

Files

Item	Description
/etc/netgroup	Contains network groups recognized by the system.
/usr/include/netdb.h	Contains the network database structures.

endprotoent Subroutine

Purpose

Closes the **/etc/protocols** file.

Library

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

Syntax

void endprotoent (void)

Description

The **endprotoent** subroutine closes the **/etc/protocols** file.

Calls made to the **getprotoent** subroutine, **getprotobyname** subroutine, or **getprotobynumber** subroutine open the **/etc/protocols** file. An application program can use the **endprotoent** subroutine to close the **/etc/protocols** file.

All applications containing the **endprotoent** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Return Values

If a previous **setprotoent** subroutine has been performed and the *StayOpen* parameter does not equal 0, the **endprotoent** subroutine will not close the **/etc/protocols** file. Also, the **setprotoent** subroutine does not indicate that it closed the file. A second **setprotoent** subroutine has to be issued with the *StayOpen* parameter equal to 0 in order for a following **endprotoent** subroutine to succeed. If this is not done, the **/etc/protocols** file closes on an **exit** subroutine.

Examples

To close the **/etc/protocols** file, type:

```
endprotoent();
```

Files

Item	Description
/etc/protocols	Contains protocol names.

Related reference
setprotoent Subroutine
getprotobyname Subroutine
getprotobynumber Subroutine
getprotoent Subroutine
getservbyport Subroutine
getservent Subroutine
setservent Subroutine
Related information
exit subroutine
Sockets Overview
Understanding Network Address Translation

endprotoent_r Subroutine

Purpose

Closes the **/etc/protocols** file.

Library

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

Syntax

```
void endprotoent_r(proto_data);
struct protoent_data *proto_data;
```

Description

The **endprotoent_r** subroutine closes the **/etc/protocols** file, which is opened by the calls made to the **getprotoent_r subroutine**, **getprotobyname_r subroutine**, or **getprotobynumber_r subroutine**.

Parameters

Item	Description
proto_data	Points to the **protoent_data** structure

Files

Item	Description
/etc/protocols	Contains protocol names.

endservent Subroutine

Purpose

Closes the **/etc/services** file.

Library

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

Syntax

```
#include <netdb.h>
```

```
void endservent ( )
```

Description

The **endservent** subroutine closes the **/etc/services** file. A call made to the **getservent** subroutine, **getservbyname** subroutine, or **getservbyport** subroutine opens the **/etc/services** file. An application program can use the **endservent** subroutine to close the **/etc/services** file.

All applications containing the **endservent** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Return Values

If a previous **setservent** subroutine has been performed and the *StayOpen* parameter does not equal 0, then the **endservent** subroutine will not close the **/etc/services** file. Also, the **setservent** subroutine does not indicate that it closed the file. A second **setservent** subroutine has to be issued with the *StayOpen* parameter equal to 0 in order for a following **endservent** subroutine to succeed. If this is not done, the **/etc/services** file closes on an **exit** subroutine.

Examples

To close the **/etc/services** file, type:

```
endservent ();
```

Files

Item	Description
/etc/services	Contains service names.

Related reference
getservbyname Subroutine
getservbyport Subroutine
Related information
exit subroutine
Sockets Overview

endservent_r Subroutine

Purpose

Closes the **/etc/services** file.

Library

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

Syntax

```
#include <netdb.h>
void endservent_r(serv_data)
struct servent_data *serv_data;
```

Description

The **endservent_r** subroutine closes the **/etc/services** file, which is opend by a call made to the **getservent_r** subroutine, **getservbyname_r** subroutine, or **getservbyport_r** subroutine opens the **/etc/services** file.

Parameters

Item	Description
serv_data	Points to the **servent_data** structure

Examples

To close the /etc/services file, type:

```
endservent_r(serv_data);
```

Files

Item	Description
/etc/services	Contains service names.

erecv, erecvmsg, erecvfrom, enrecvmsg, or enrecvfrom Subroutine

Purpose

Allows applications to receive messages from sockets along with the Sensitivity Level (SL).

Library

The libraries that are available in the **erecv** subroutines are:

1. Standard C Library (**libc.a**)
2. Trusted AIX Sensitivity Label Library (**libmls.a**)

Syntax

```
#include <sys/socket.h>
#include <sys/mac.h>
```

```
int erecv (Socket, Buffer, Length, Flags, Label)
int  Socket;
void * Buffer;
size_t  Length;
int  Flags;
sec_labels_t *Label;

int erecvmsg ( Socket,  Message,  Flags, Label)
int Socket;
struct msghdr Message [ ];
int Flags;
sec_labels_t *Label;

ssize_t erecvfrom (Socket, Buffer, Length, Flags, From, FromLength, Label)
int  Socket;
void * Buffer;
size_t  Length,
int  Flags;
struct sockaddr * From;
socklen_t * FromLength;
sec_labels_t *Label;

int enrecvmsg (Socket, Message, Flags, Label)
int Socket;
struct msghdr Message [ ];
int Flags;
sec_labels_t *Label;

ssize_t enrecvfrom (Socket, Buffer, Length, Flags, From, FromLength, Label)
int Socket;
void *Buffer;
size_t Length;
int Flags;
struct sockaddr *From;
socklen_t *FromLength;
sec_labels_t *Label;
```

Description

The **erecv**, **erecvmsg**, **erecvfrom**, **enrecvmsg**, and **enrecvfrom** subroutines work exactly like the **recv**, **recvmsg**, **recvfrom**, **nrecvmsg**, and **nrecvfrom** subroutines respectively, except that the **erecv**, **erecvmsg**, **erecvfrom**, **enrecvmsg**, and **enrecvfrom** subroutines allow the application to retrieve the SL from the received data by providing a valid *Label* parameter.

If no messages are available at the socket, the **erecv**, **erecvmsg**, **erecvfrom**, **enrecvmsg**, and **enrecvfrom** subroutines wait for a message to arrive, unless the socket is nonblocking. If a socket is nonblocking, the system returns an error.

Parameters

Item	Description
Socket	Specifies the socket descriptor.
Buffer	Specifies the address where the message is placed.
Length	Specifies the size of the *Buffer* parameter.

Item	Description
Flags	Points to a value controlling the message reception. The **/usr/include/sys/socket.h** file defines the *Flags* parameter. The argument to receive a call is formed by the logical OR operation with one or more of the following values: **MSG_OOB** Processes out-of-band data. The significance of out-of-band data is protocol dependent. **MSG_PEEK** Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to the **erecv**, **erecvmsg**, **erecvfrom**, **enrecvmsg**, or **enrecvfrom** subroutine or a similar subroutine. **MSG_WAITALL** Requests that the subroutine does not return until the requested number of bytes are read. The subroutine can return fewer bytes than the requested number if a signal is caught, the connection is terminated, or an error is pending for the socket. The subroutine can also return fewer bytes when the SL information across the data stream is different. Only those bytes that have the same SL information are returned to the user.
Message	Points to the address of the **msghdr** structure, which contains both the address for the incoming message and the space for the sender address.
From	Points to a socket structure, containing the address of the source.
FromLength	Specifies the length of the address of the sender or of the source.
Label	Specifies a result parameter that contains the SL from the received data.

Return Values

Upon successful completion, the subroutines return the length of the message in bytes.

When an error occurs, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Returns a value of 0 if the connection disconnects (in case of connected sockets).
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **erecv**, **erecvmsg**, **erecvfrom**,**enrecvmsg**, or **enrecvfrom** subroutine is unsuccessful if any of the following errors occurs:

Item	Description
EBADF	The *Socket* parameter is not valid.
ECONNRESET	The remote peer forced the connection to be closed.
EFAULT	The data was directed into a nonexistent or protected part of the process address space. (The *Buffer* parameter is not valid.)
EINTR	A signal interrupted the **erecv**, **erecvmsg**, **erecvfrom**, **enrecvmsg**, or **enrecvfrom** subroutine before any data is available.
EINVAL	The **MSG_OOB** value was set and no out-of-band data was available.
ENOBUF	Insufficient resources are available in the system to perform the operation.
ENOTCONN	A receiving operation was attempted on a **SOCK_STREAM** socket that was not connected.

Item	Description
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The **MSG_OOB** value is set for a **SOCK_DGRAM** socket or any **AF_UNIX** socket.
ETIMEDOUT	The connection timed out during connection establishment, or there was a transmission timeout on an active connection.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.
EACCES	The MLS MAC check failed.

esend, esendto, or esendmsg Subroutine

Purpose

Allows an application to send messages on a socket with the Sensitivity Level (SL) different from that of its own.

Library

Standard C Library (**libc.a**)Trusted AIX Sensitivity Label Library (**libmls.a**)

Syntax

```
#include <sys/types.h>
#include <sys/socketvar.h>
#include <sys/mac.h>
#include <sys/socket.h>

int esend (Socket, Message, Length, Flags, Label)
int  Socket;
const void * Message;
size_t  Length;
int  Flags;
sec_labels_t *Label;

int esendmsg ( Socket,  Message,  Flags, Label)
int Socket;
const struct msghdr Message [ ];
int Flags;
sec_labels_t *Label;

int esendto (Socket, Message, Length, Flags, To, ToLength, Label)
int  Socket;
const void * Message;
size_t  Length;
int  Flags;
const struct sockaddr * To;
socklen_t  ToLength;
sec_labels_t *Label;
```

Description

The **esend**, **esendmsg**, and **esendto** subroutines work exactly like **send**, **sendmsg** and **sendto** subroutines respectively, except that the **esend**, **esendmsg**, and **esendto** subroutines allow applications to associate a Sensitivity Level different from their own to the outgoing data through the *Label* parameter.

The **esend** subroutine can be used on connected sockets only. The **esendto** and **esendmsg** subroutines can be used with connected or unconnected sockets.

For **SOCK_STREAM** socket types, when the SL is changed between subsequent send operations, the application is blocked until the pending data on the socket buffer can be flushed. If the socket is marked as nonblocking type and there is pending data on the socket buffer, an error is returned.

Parameters

Item	Description
Socket	Specifies a unique name for the socket.
Message	Points to the address of the message or the **msghdr** structure containing the message to send.
Length	Specifies the length of the message in bytes.
Flags	Allows the sender to control the transmission of the message. **MSG_OOB** Processes out-of-band data on sockets that support **SOCK_STREAM** communication. **MSG_DONTROUTE** Sends without using routing tables. **MSG_MPEG2** Indicates that this block is a MPEG2 block. This value is valid **SOCK_CONN_DGRAM** socket types only.
To	Specifies the destination address for the message. The destination address is a **sockaddr** structure defined in the **/usr/include/sys/socket.h** file.
ToLength	Specifies the size of the destination address.
Label	Specifies the SL to be used on the outgoing data.

Return Values

Upon successful completion, the **esend**, **esendmsg**, or **esendto** subroutine returns the number of characters sent.

If errors occur, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **esend**, **esendmsg**, or **esendto** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EACCES	Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability, or the MLS MAC check failed.
EADDRNOTAVAIL	The specified address is not valid.
EAFNOSUPPORT	The specified address is not a valid address for the address family of this socket.
EBADF	The *Socket* parameter is not valid.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not in connection mode and no peer address is set.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EHOSTUNREACH	The destination host cannot be reached.
EINTR	A signal interrupted the **esend**, **esendmsg**, or **esendto** subroutine before any data was transmitted.

Error	Description
EINVAL	The *Length* parameter is not valid.
EISCONN	A **SOCK_DGRAM** socket is already connected.
EMSGSIZE	The message is too large to be sent all at once, as the socket requires.
ENETUNREACH	The destination network is not reachable.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
ENOENT	The path name does not contain an existing file, or the path name is an empty string.
ENOMEM	The available data space in memory is not large enough to hold group or ACL information.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The *Socket* parameter is associated with a socket that does not support one or more of the values set in the *Flags* parameter.
EPIPE	An attempt was made to send on a socket that was connected, but the connection was shut down either by the remote peer or by this side of the connection. If the socket is of type **SOCK_STREAM**, the **SIGPIPE** signal is generated for the calling process.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted. Or a sending operation was attempted with different SLs while there was pending data on the socket buffer, and the socket was marked nonblocking

ether_ntoa, ether_aton, ether_ntohost, ether_hostton, or ether_line Subroutine

Purpose

Maps 48-bit Ethernet numbers.

Library

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

Syntax

```
#include <arap/inet.h>
```

```
char *ether_ntoa (EthernetNumber)
struct ether_addr * EthernetNumber;

struct ether_addr *ether_aton( String);
char *string

int *ether_ntohost (HostName, EthernetNumber)
char * HostName;
struct ether_addr *EthernetNumber;
```

```
int *ether_hostton (HostName, EthernetNumber)
char *HostName;
struct ether_addr *EthernetNumber;
```

```
int *ether_line (Line, EthernetNumber, HostName)
char * Line, *HostName;
struct ether_addr *EthernetNumber;
```

Description

> **Attention:** Do not use the **ether_ntoa** or **ether_aton** subroutine in a multithreaded environment.

The **ether_ntoa** subroutine maps a 48-bit Ethernet number pointed to by the *EthernetNumber* parameter to its standard ASCII representation. The subroutine returns a pointer to the ASCII string. The representation is in the form *x***:***x***:***x***:***x***:***x***:***x***:** where *x* is a hexadecimal number between 0 and ff. The **ether_aton** subroutine converts the ASCII string pointed to by the *String* parameter to a 48-bit Ethernet number. This subroutine returns a null value if the string cannot be scanned correctly.

The **ether_ntohost** subroutine maps a 48-bit Ethernet number pointed to by the *EthernetNumber* parameter to its associated host name. The string pointed to by the *HostName* parameter must be long enough to hold the host name and a null character. The **ether_hostton** subroutine maps the host name string pointed to by the *HostName* parameter to its corresponding 48-bit Ethernet number. This subroutine modifies the Ethernet number pointed to by the *EthernetNumber* parameter.

The **ether_line** subroutine scans the line pointed to by *line* and sets the hostname pointed to by the *HostName* parameter and the Ethernet number pointed to by the *EthernetNumber* parameter to the information parsed from *LINE*.

Parameters

Item	Description
EthernetNumber	Points to an Ethernet number.
String	Points to an ASCII string.
HostName	Points to a host name.
Line	Points to a line.

Return Values

Item	Description
0	Indicates that the subroutine was successful.
non-zero	Indicates that the subroutine was not successful.

Files

Item	Description
/etc/ethers	Contains information about the known (48-bit) Ethernet addresses of hosts on the Internet.

Related information
Subroutines Overview
List of Multithread Subroutines

f

AIX runtime services beginning with the letter *f*.

FrcaCacheCreate Subroutine

Purpose

Creates a cache instance within the scope of a Fast Response Cache Accelerator (FRCA) instance.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCacheCreate ( CacheHandle,  FrcaHandle,  CacheSpec);
int32_t *  CacheHandle;
int32_t    FrcaHandle;
frca_cache_create_t *  CacheSpec;
```

Description

The **FrcaCacheCreate** subroutine creates a cache instance for an FRCA instance that has already been configured. Multiple caches can be created for an FRCA instance. Cache handles are unique only within the scope of the FRCA instance.

Parameters

Item	Description
CacheHandle	Returns a handle that is required by the other cache-related subroutines of the FRCA API to refer to the newly created FRCA cache instance.
FrcaHandle	Identifies the FRCA instance for which the cache is created.
CacheSpec	Points to a **frca_ctrl_create_t** structure, which specifies the characteristics of the cache to be created. The structure contains the following members: **uint32_t** *cacheType*; **uint32_t** *nMaxEntries*; **Note:** Structure members do not necessarily appear in this order. ***cacheType*** Specifies the type of the cache instance. This field must be set to FCTRL_SERVERTYPE_HTTP. ***nMaxEntries*** Specifies the maximum number of entries allowed for the cache instance.

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
EINVAL	The *CacheHandle* or the *CacheSpec* parameter is zero or the *CacheSpec* parameter is not of the correct type **FCTRL_CACHETYPE_HTTP**.
EFAULT	The *CacheHandle* or the *CacheSpec* point to an invalid address.

Item	Description
ENOENT	The *FrcaHandle* parameter is invalid.

FrcaCacheDelete Subroutine

Purpose

Deletes a cache instance within the scope of a Fast Response Cache Accelerator (FRCA) instance.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCacheDelete ( CacheHandle,  FrcaHandle);
int32_t  CacheHandle;
int32_t  FrcaHandle;
```

Description

The **FrcaCacheDelete** subroutine deletes a cache instance and releases any associated resources.

Parameters

Item	Description
CacheHandle	Identifies the cache instance that is to be deleted.
FrcaHandle	Identifies the FRCA instance to which the cache instance belongs.

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
ENOENT	The *CacheHandle* or the *FrcaHandle* parameter is invalid.

FrcaCacheLoadFile Subroutine

Purpose

Loads a file into a cache associated with a Fast Response Cache Accelerator (FRCA) instance.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCacheLoadFile ( CacheHandle,  FrcaHandle,  FileSpec,  AssocData);
int32_t  CacheHandle;
int32_t  FrcaHandle;
frca_filespec_t * FileSpec;
frca_assocdata_t * AssocData;
```

Description

The **FrcaCacheLoadFile** subroutine loads a file into an existing cache instance for an previously configured FRCA instance.

Parameters

Item	Description
CacheHandle	Identifies the cache instance to which the new entry should be added.
FrcaHandle	Identifies the FRCA instance to which the cache instance belongs.
FileSpec	Points to a **frca_loadfile_t** structure, which specifies characteristics used to identify the cache entry that is to be loaded into the given cache. The structure contains the following members: **uint32_t** *cacheEntryType*; **char *** *fileName*; **char *** *virtualHost*; **char *** *searchKey*; **Note:** Structure members do not necessarily appear in this order. ***cacheEntryType*** Specifies the type of the cache entry. This field must be set to FCTRL_CET_HTTPFILE. ***fileName*** Specifies the absolute path to the file that is providing the contents for the new cache entry. ***virtualHost*** Specifies a virtual host name that is being served by the FRCA instance. ***searchKey*** Specifies the key that the cache entry can be found under by the FRCA instance when it processes an intercepted request. For the HTTP GET engine, the search key is identical to the *abs_path* part of the HTTP URL according to section 3.2.2 of RFC 2616. For example, the search key corresponding to the URL http://www.mydomain/welcome.html is /welcome.html. **Note:** If a cache entry with the same type, file name, virtual host, and search key already exists and the file has not been modified since the existing entry was created, the load request succeeds without any effect. If the entry exists and the file's contents have been modified since being loaded into the cache, the cache entry is updated. If the entry exists and the file's contents have not changed, but any of the settings of the HTTP header fields change, the existing entry must be unloaded first.

Item	Description
AssocData	Points to a **frca_assocdata_t** structure, which specifies additional information to be associated with the contents of the given cache entry. The structure contains the following members: **uint32_t** *assocDataType*; **char *** *cacheControl*; **char *** *contentType*; **char *** *contentEncoding*; **char *** *contentLanguage*; **char *** *contentCharset*; **Note:** Structure members do not necessarily appear in this order. ***assocDataType*** Specifies the type of data that is associated with the given cache entry. ***cacheControl*** Specifies the settings of the corresponding HTTP header field according to RFC 2616. ***contentType*** Specifies the settings of the corresponding HTTP header field according to RFC 2616. ***contentEncoding*** Specifies the settings of the corresponding HTTP header field according to RFC 2616. ***contentLanguage*** Specifies the settings of the corresponding HTTP header field according to RFC 2616. ***contentCharset*** Specifies the settings of the corresponding HTTP header field according to RFC 2616.

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
EINVAL	The *FileSpec* or the *AssocData* parameter is zero or are not of the correct type or any of the *fileName* or the *searchKey* components are zero or the size of the file is zero.
EFAULT	The *FileSpec* or the *AssocData* parameter or one of their components points to an invalid address.
ENOMEM	The FRCA or NBC subsystem is out of memory.
EFBIG	The content of the cache entry failed to load into the NBC. Check network options **nbc_limit**, **nbc_min_cache**, and **nbc_max_cache**.

Item	Description
ENOTREADY	The kernel extension is currently being loaded or unloaded.
ENOENT	The *CacheHandle* or the *FrcaHandle* parameter is invalid.

FrcaCacheUnloadFile Subroutine

Purpose

Removes a cache entry from a cache that is associated with a Fast Response Cache Accelerator (FRCA) instance.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCacheUnoadFile ( CacheHandle,  FrcaHandle,  FileSpec);
int32_t  CacheHandle;
int32_t  FrcaHandle;
frca_filespec_t * FileSpec;
```

Description

The **FrcaCacheUnoadFile** subroutine removes a cache entry from an existing cache instance for an previously configured FRCA instance.

Parameters

Item	Description
CacheHandle	Identifies the cache instance from which the entry should be removed.
FrcaHandle	Identifies the FRCA instance to which the cache instance belongs.

Item	Description
FileSpec	Points to a **frca_loadfile_t** structure, which specifies characteristics used to identify the cache entry that is to be removed from the given cache. The structure contains the following members: **uint32_t** *cacheEntryType*; **char *** *fileName*; **char *** *virtualHost*; **char *** *searchKey*; **Note:** Structure members do not necessarily appear in this order. ***cacheEntryType*** Specifies the type of the cache entry. This field must be set to FCTRL_CET_HTTPFILE. ***fileName*** Specifies the absolute path to the file that is to be removed from the cache. ***virtualHost*** Specifies a virtual host name that is being served by the FRCA instance. ***searchKey*** Specifies the key under which the cache entry can be found.

Note: The **FrcaCacheUnoadFile** subroutine succeeds if a cache entry with the same type, file name, virtual host, and search key does not exist. This subroutine fails if the file associated with *fileName* does not exist or if the calling process does not have sufficient access permissions.

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
EINVAL	The *FileSpec* parameter is zero or the *cacheEntryType* component is not set to **FCTRL_CET_HTTPFILE** or the *searchKey* component is zero or the *fileName* is '/' or the *fileName* is not an absolute path.
EFAULT	The *FileSpec* parameter or one of the components points to an invalid address.
EACCES	Access permission is denied on the *fileName*.

FrcaCtrlCreate Subroutine

Purpose

Creates a Fast Response Cache Accelerator (FRCA) control instance.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCtrlCreate ( FrcaHandle,  InstanceSpec);
int32_t * FrcaHandle;
frca_ctrl_create_t * InstanceSpec;
```

Description

The **FrcaCtrlCreate** subroutine creates and configures an FRCA instance that is associated with a previously configured TCP listen socket. TCP connections derived from the TCP listen socket are intercepted by the FRCA instance and, if applicable, adequate responses are generated by the in-kernel code on behalf of the user-level application.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters

Item	Description
FrcaHandle	Returns a handle that is required by the other FRCA API subroutines to refer to the newly configured FRCA instance.

Item	Description
InstanceSpec	Points to a **frca_ctrl_create_t** structure, which specifies the parameters used to configure the newly created FRCA instance. The structure contains the following members:

```
uint32_t  serverType;
char * serverName;
uint32_t  nListenSockets;
uint32_t * ListenSockets;
uint32_t  flags;
uint32_t  nMaxConnections;
uint32_t  nLogBufs;
char * logFile;
```

Note: Structure members do not necessarily appear in this order.

serverType
Specifies the type for the FRCA instance. This field must be set to FCTRL_SERVERTYPE_HTTP.

serverName
Specifies the value to which the HTTP header field is set.

nListenSocket
Specifies the number of listen socket descriptors pointed to by *listenSockets*.

listenSocket
Specifies the TCP listen socket that the FRCA instance should be configured to intercept.

Note: The TCP listen socket must exist and the SO_KERNACCEPT socket option must be set at the time of calling the **FrcaCtrlCreate** subroutine.

flags
Specifies the logging format, the initial state of the logging subsystem, and whether responses generated by the FRCA instance should include the **Server:** HTTP header field. The valid flags are as follows:

FCTRL_KEEPALIVE

FCTRL_LOGFORMAT

FCTRL_LOGFORMAT_ECLF

FCTRL_LOGFORMAT_VHOST

FCTRL_LOGMODE

FCTRL_LOGMODE_ON

FCTRL_SENDSERVERHEADER

nMaxConnections
Specifies the maximum number of intercepted connections that are allowed at any given point in time.

nLogBufs
Specifies the number of preallocated logging buffers used for logging information about HTTP GET requests that have been served successfully.

Item	Description
	logFile Specifies the absolute path to a file used for appending logging information. The HTTP GET engine uses *logFile* as a base name and appends a sequence number to it to generate the actual file name. Whenever the size of the current log file exceeds the threshold of approximately 1 gigabyte, the sequence number is incremented by 1 and the logging subsystem starts appending to the new log file. **Note:** The FRCA instance creates the log file, but not the path to it. If the path does not exist or is not accessible, the FRCA instance reverts to the default log file **/tmp/frca.log**.

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
EINVAL	The *FrcaHandle* or the *InstanceSpec* parameter is zero or is not of the correct type or the *listenSockets* components do not specify any socket descriptors.
EFAULT	The *FrcaHandle* or the *InstanceSpec* or a component of the *InstanceSpec* points to an invalid address.
ENOTREADY	The kernel extension is currently being loaded or unloaded.
ENOTSOCK	A TCP listen socket does not exist.

FrcaCtrlDelete Subroutine

Purpose

Deletes a Fast Response Cache Accelerator (FRCA) control instance.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCtrlDelete ( FrcaHandle);
int32_t * FrcaHandle;
```

Description

The **FrcaCtrlDelete** subroutine deletes an FRCA instance and releases any associated resources.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters

Item	Description
FrcaHandle	Identifies the FRCA instance on which this operation is performed.

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
ENOENT	The *FrcaHandle* parameter is invalid.
ENOTREADY	The FRCA control instance is in an undefined state.

FrcaCtrlLog Subroutine

Purpose

Modifies the behavior of the logging subsystem.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCtrlLog ( FrcaHandle,  Flags);
int32_t  FrcaHandle;
uint32_t  Flags;
```

Description

The **FrcaCtrlLog** subroutine modifies the behavior of the logging subsystem for the Fast Response Cache Accelerator (FRCA) instance specified. Modifiable attributes are the logging mode, which can be turned on or off, and the logging format, which defaults to the HTTP Common Log Format (CLF). The logging format can be changed to Extended Common Log Format (ECLF) and can be set to include virtual host information.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters

Item	Description
FrcaHandle	Returns a handle that is required by the other FRCA API subroutines to refer to the newly configured FRCA instance.
Flags	Specifies the behavior of the logging subsystem. The parameter value is constructed by logically ORing single flags. The valid flags are as follows: **FCTRL_LOGFORMAT** **FCTRL_LOGFORMAT_ECLF** **FCTRL_LOGFORMAT_VHOST** **FCTRL_LOGMODE** **FCTRL_LOGMODE_ON**

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
ENOTREADY	The kernel extension is currently being loaded or unloaded.

FrcaCtrlStart Subroutine

Purpose

Starts the interception of TCP data connections for a previously configured Fast Response Cache Accelerator (FRCA) instance.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCtrlStart ( FrcaHandle);
int32_t * FrcaHandle;
```

Description

The **FrcaCtrlStart** subroutine starts the interception of TCP data connections for an FRCA instance. If the FRCA instance cannot handle the data on that connection, it passes the data to the user-level application that has established the listen socket.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters

Item	Description
FrcaHandle	Identifies the FRCA instance on which this operation is performed.

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
ENOENT	The *FrcaHandle* parameter is invalid.
ENOTREADY	The FRCA control instance is in an undefined state.
ENOTSOCK	A TCP listen socket that was passed in with the *FrcaCtrlCreate* cannot be intercepted because it does not exist.

FrcaCtrlStop Subroutine

Purpose

Stops the interception of TCP data connections for a Fast Response Cache Accelerator (FRCA) instance.

Library

FRCA Library (**libfrca.a**)

Syntax

```
#include <frca.h>
int32_t FrcaCtrlStop ( FrcaHandle);
int32_t * FrcaHandle;
```

Description

The **FrcaCtrlStop** subroutine stops the interception of newly arriving TCP data connections for a previously configured FRCA instance. Connection requests are passed to the user-level application that has established the listen socket.

The only FRCA instance type that is currently supported handles static GET requests as part of the Hypertext Transfer Protocol (HTTP).

Parameters

Item	Description
FrcaHandle	Identifies the FRCA instance on which this operation is performed.

Return Values

Item	Description
0	The subroutine completed successfully.
-1	The subroutine failed. The global variable *errno* is set to indicate the specific type of error.

Error Codes

Item	Description
ENOENT	The *FrcaHandle* parameter is invalid.
ENOTREADY	The FRCA control instance has not been started yet.

freeaddrinfo Subroutine

Purpose

Frees memory allocated by the “getaddrinfo Subroutine” on page 144.

Library

The Standard C Library (<**libc.a**>)

Syntax

```
#include <sys/socket.h>
#include <netdb.h>
void freeaddrinfo (struct addrinfo *ai)
```

Description

The **freeaddrinfo** subroutine frees one or more **addrinfo** structures returned by the **getaddrinfo** subroutine, along with any additional storage associated with those structures. If the **ai_next** field of the structure is not NULL, the entire list of structures is freed.

Parameters

Item	Description
ai	Points to dynamic storage allocated by the **getaddrinfo** subroutine

Related information

gai_strerror Subroutine

g

AIX runtime services beginning with the letter *g*.

getaddrinfo Subroutine

Purpose

Protocol-independent hostname-to-address translation.

Library

Library (**libc.a**)

Syntax

```
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo (hostname, servname, hints, res)
const char *hostname;
const char *servname;
const struct addrinfo *hints;
struct addrinfo **res;
```

Description

The *hostname* and *servname* parameters describe the hostname and/or service name to be referenced. Zero or one of these arguments may be NULL. A non-NULL hostname may be either a hostname or a numeric host address string (a dotted-decimal for IPv4 or hex for IPv6). A non-NULL servname may be either a service name or a decimal port number.

The *hints* parameter specifies hints concerning the desired return information. The *hostname* and *servname* parameters are pointers to null-terminated strings or NULL. One or both of these arguments must be a non-NULL pointer. In a normal client scenario, both the *hostname* and *servname* parameters are specified. In the normal server scenario, only the *servname* parameter is specified. A non-NULL hostname string can be either a host name or a numeric host address string (for example, a dotted-decimal IPv4 address or an IPv6 hex address). A non-NULL *servname* string can be either a service name or a decimal port number.

The caller can optionally pass an **addrinfo** structure, pointed to by the *hints* parameter, to provide hints concerning the type of socket that the caller supports. In this **hints** structure, all members other than **ai_flags**, **ai_eflags ai_family**, **ai_socktype**, and **ai_protocol** must be zero or a NULL pointer. A value of PF_UNSPEC for **ai_family** means the caller will accept any protocol family. A value of zero for **ai_socktype** means the caller accepts any socket type. A value of zero for **ai_protocol** means the caller accepts any protocol. For example, if the caller handles only TCP and not UDP, the **ai_socktype** member of the **hints** structure should be set to SOCK_STREAM when the **getaddrinfo** subroutine is called. If the caller handles only IPv4 and not IPv6, the **ai_family** member of the **hints** structure should be set to PF_INET when **getaddrinfo** is called. If the *hints* parameter in **getaddrinfo** is a NULL pointer, it is the same as if the caller fills in an **addrinfo** structure initialized to zero with **ai_family** set to PF_UNSPEC.

Upon successful return, a pointer to a linked list of one or more **addrinfo** structures is returned through the *res* parameter. The caller can process each **addrinfo** structure in this list by following the **ai_next** pointer, until a NULL pointer is encountered. In each returned **addrinfo** structure the three members **ai_family**, **ai_socktype**, and **ai_protocol** are the corresponding arguments for a call to the **socket** subroutine. In each **addrinfo** structure, the **ai_addr** member points to a filled-in socket address structure whose length is specified by the **ai_addrlen** member.

If the AI_PASSIVE bit is set in the **ai_flags** member of the **hints** structure, the caller plans to use the returned socket address structure in a call to the **bind** subroutine. If the *hostname* parameter is a NULL pointer, the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address.

If the AI_PASSIVE bit is not set in the **ai_flags** member of the hints structure, the returned socket address structure is ready for a call to the **connect** subroutine (for a connection-oriented protocol) or the **connect**, **sendto**, or **sendmsg** subroutine (for a connectionless protocol). If the *hostname* parameter is a NULL pointer, the IP address portion of the socket address structure is set to the loopback address.

If the AI_CANONNAME bit is set in the **ai_flags** member of the hints structure, upon successful return the **ai_canonname** member of the first **addrinfo** structure in the linked list points to a NULL-terminated string containing the canonical name of the specified hostname.

If the AI_NUMERICHOST flag is specified, a non-NULL nodename string supplied is a numeric host address string. Otherwise, an (EAI_NONAME) error is returned. This flag prevents any type of name resolution service (such as, DNS) from being invoked.

If the AI_NUMERICSERV flag is specified, a non-NULL servname string supplied is a numeric port string. Otherwise, an (EAI_NONAME) error is returned. This flag prevents any type of name resolution service (such as, NIS+) from being invoked.

If the AI_V4MAPPED flag is specified along with an **ai_family** value of AF_INET6, the **getaddrinfo** subroutine returns IPv4-mapped IPv6 addresses when no matching IPv6 addresses (**ai_addrlen** is 16) are found. For example, when using DNS, if no AAAA or A6 records are found, a query is made for A records. Any found are returned as IPv4-mapped IPv6 addresses. The AI_V4MAPPED flag is ignored unless **ai_family** equals AF_INET6.

If the AI_ALL flag is used with the AI_V4MAPPED flag, the **getaddrinfo** subroutine returns all matching IPv6 and IPv4 addresses. For example, when using DNS, a query is first made for AAAA/A6 records. If successful, those IPv6 addresses are returned. Another query is made for A records, and any IPv4 addresses found are returned as IPv4-mapped IPv6 addresses. The AI_ALL flag without the AI_V4MAPPED flag is ignored.

Note: When **ai_family** is not specified (AF_UNSPEC), AI_V4MAPPED and AI_ALL flags are used if AF_INET6 is supported.

If the AI_EXTFLAGS is specified in the **ai_flags** member of the hints structure and **ai_eflags** is specified as a non zero value, the address selection algorithm is affected. The address selection algorithm orders the list of returned addrinfo structures using a set of ordered rules (RFC 3484) taking into account the address contained in the ai_addr member of each addrinfo structure and the source addresses from which this address can be reached. The ai_eflags expresses preferences meaning that the rules described below will be applied if a higher rule has not ordered the set of addresses before.

The **ai_eflags** can be set to a combination of the following flags:

- IPV6_PREFER_SRC_HOME: prefer addresses reachable from a Home source address
- IPV6_PREFER_SRC_COA: prefer addresses reachable from a Care-of source address
- IPV6_PREFER_SRC_TMP: prefer addresses reachable from a temporary address
- IPV6_PREFER_SRC_PUBLIC: the prefer addresses reachable from a public source address
- IPV6_PREFER_SRC_CGA: the prefer addresses reachable from a Cryptographically Generated Address (CGA) source address
- IPV6_PREFER_SRC_NONCGA: the prefer addresses reachable from a non-CGA source address

For instance, the IPV6_PREFER_SRC_TMP ai_eflags means that the address selection algorithm will order the returned addrinfo structures with addresses reachable from a temporary address before the ones with addresses reachable from a public address whenever possible. Setting contradictory flags (e.g. IPV6_PREFER_SRC_TMP and IPV6_PREFER_SRC_PUBLIC) at the same time results in the error EINVAL.

If the AI_ADDRCONFIG flag is specified, a query for AAAA or A6 records should occur only if the node has at least one IPv6 source address configured. A query for A records should occur only if the node has at least one IPv4 source address configured. The loopback address is not considered valid as a configured source address.

All of the information returned by the **getaddrinfo** subroutine is dynamically allocated: the **addrinfo** structures, the socket address structures, and canonical host name strings pointed to by the **addrinfo** structures. To return this information to the system, freeaddrinfo subroutine is called.

The addrinfo structure is defined as:

```
struct addrinfo {
  int               ai_flags;          /* AI_PASSIVE, AI_CANONNAME */
  int               ai_family;         /* PF_xxx */
  int               ai_socktype;       /* SOCK_xxx */
  int               ai_protocol;       /* 0 or IP=PROTO_xxx for IPv4 and IPv6 */
  size_t            ai_addrlen;        /* length of ai_addr */
  char              *ai_canonname;     /* canoncial name for hostname */
  struct sockaddr   *ai_addr;          /* binary address */
```

```
  struct addrinfo   *ai_next;          /* next structure in linked list */
  int ai_eflags; /* Extended flags for special usage */
}
```

Return Values

If the query is successful, a pointer to a linked list of one or more **addrinfo** structures is returned via the *res* parameter. A zero return value indicates success. If the query fails, a non-zero error code is returned.

Error Codes

The following names are the non-zero error codes. See *netdb.h* for further definition.

Item	Description
EAI_ADDRFAMILY	Address family for hostname not supported
EAI_AGAIN	Temporary failure in name resolution
EAI_BADFLAGS	Invalid value for **ai_flags**
EAI_FAIL	Non-recoverable failure in name resolution
EAI_FAMILY	**ai_family** not supported
EAI_MEMORY	Memory allocation failure
EAI_NODATA	No address associated with *hostname*
EAI_NONAME	No *hostname* nor *servname* provided, or not known
EAI_SERVICE	*servname* not supported for **ai_socktype**
EAI_SOCKTYPE	**ai_socktype** not supported
EAI_SYSTEM	System error returned in errno
EAI_BADEXTFLAGS	Invalid value for **ai_eflags.**

Related information
gai_strerror Subroutine

get_auth_method Subroutine

Purpose

Returns the list of authentication methods for the secure rcmds.

Library

Authentication Methods Library (**libauthm.a**)

Syntax

Description

This method returns the authentication methods currently configured in the order in which they should be attempted in the unsigned integer pointer the user passed in.

The list in the unsigned integer pointer is either NULL (on an error) or is an array of unsigned integers terminated by a zero. Each integer identifies an authentication method. The order that a client should attempt to authenticate is defined by the order of the list.

Note: The calling routine is responsible for freeing the memory in which the list is contained.

The flags identifying the authentication methods are defined in the **/usr/include/authm.h** file.

Parameter

Item	Description
authm	Points to an array of unsigned integers. The list of authentication methods is returned in the zero terminated list.

Return Values

Upon successful completion, the **get_auth_method** subroutine returns a zero.

Upon unsuccessful completion, the **get_auth_method** subroutine returns an **errno**.

getdomainname Subroutine

Purpose

Gets the name of the current domain.

Library

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

Syntax

```
int getdomainname ( Name,  Namelen)
char *Name;
int Namelen;
```

Description

The **getdomainname** subroutine returns the name of the domain for the current processor as previously set by the **setdomainname** subroutine. The returned name is null-terminated unless insufficient space is provided.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. Only the Network Information Service (NIS) and the **sendmail** command make use of domains.

All applications containing the **getdomainname** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Note: Domain names are restricted to 256 characters.

Parameters

Item	Description
Name	Specifies the domain name to be returned.
Namelen	Specifies the size of the array pointed to by the *Name* parameter.

Return Values

If the call succeeds, a value of 0 is returned. If the call is unsuccessful, a value of -1 is returned and an error code is placed in the **errno** global variable.

Error Codes

The following error may be returned by this subroutine:

Value	Description
EFAULT	The *Name* parameter gave an invalid address.

Related reference
setdomainname Subroutine
Related information
Sockets Overview

gethostbyaddr Subroutine

Purpose

Gets network host entry by address.

Library

```
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>
```

```
struct hostent *gethostbyaddr ( Address,  Length,  Type)
const void *Address, size_t Length, int Type;
```

Description

The **gethostbyaddr** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **gethostbyaddr** subroutine retrieves information about a host using the host address as a search key. Unless specified, the **gethostbyaddr** subroutine uses the default name services ordering, that is, it will query DNS/BIND, NIS, then the local **/etc/hosts** file.

When using DNS/BIND name service resolution, if the file **/etc/resolv.conf** exists, the **gethostbyaddr** subroutine queries the domain name server. The **gethostbyaddr** subroutine recognizes domain name servers as described in RFC 883.

When using NIS for name resolution, if the **getdomainname** subroutine is successful and **yp_bind** indicates NIS is running, then the **gethostbyaddr** subroutine queries NIS.

The **gethostbyaddr** subroutine also searches the local **/etc/hosts** file when indicated to do so.

The **gethostbyaddr** returns a pointer to a **hostent** structure, which contains information obtained from one of the name resolutions services. The **hostent** structure is defined in the **netdb.h** file.

The environment variable, NSORDER can be set to override the default name services ordering and the order specified in the **/etc/netsvc.conf** file.

Parameters

Item	Description
Address	Specifies a host address. The host address is passed as a pointer to the binary format address.
Length	Specifies the length of host address.
Type	Specifies the domain type of the host address. It can be either AF_INET or AF_INET6.

Return Values

The **gethostbyaddr** subroutine returns a pointer to a **hostent** structure upon success.

If an error occurs or if the end of the file is reached, the **gethostbyaddr** subroutine returns a NULL pointer and sets **h_errno** to indicate the error.

Error Codes

The **gethostbyaddr** subroutine is unsuccessful if any of the following errors occur:

Error	Description
HOST_NOT_FOUND	The host specified by the *Name* parameter is not found.
TRY_AGAIN	The local server does not receive a response from an authoritative server. Try again later.
NO_RECOVERY	This error code indicates an unrecoverable error.
NO_ADDRESS	The requested *Address* parameter is valid but does not have a name at the name server.
SERVICE_UNAVAILABLE	None of the name services specified are running or available.

Files

Item	Description
/etc/hosts	Contains the host-name database.
/etc/resolv.conf	Contains the name server and domain name information.
/etc/netsvc.conf	Contains the name of the services ordering.
/usr/include/netdb.h	Contains the network database structure.

Related reference
gethostbyname Subroutine
Related information
Sockets Overview
Network Address Translation

gethostbyaddr_r Subroutine

Purpose

Gets network host entry by address.

Library

```
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>
int gethostbyadd_r(Addr, Len, Type, Htent, Ht_data)
const char *Addr, size_t Len, int Type, struct hostent *Htent, struct hostent_data *Ht_data;
```

Description

This function internally calls the **gethostbyaddr** subroutine and stores the value returned by the **gethostbyaddr** subroutine to the hostent structure.

Parameters

Item	Description
Addr	Points to the host address that is a pointer to the binary format address.
Len	Specifies the length of the address.
Type	Specifies the domain type of the host address. It can be either AF_INET or AF_INET6.
Htent	Points to a hostent structure which is used to store the return value of the **gethostaddr** subroutine.
Ht_data	Points to a **hostent_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Note: The return value of the **gethostbyaddr** subroutine points to static data that is overwritten by subsequent calls. This data must be copied at every call to be saved for use by subsequent calls. The **gethostbyaddr_r** subroutine solves this problem.

If the *Name* parameter is a **hostname**, this subroutine searches for a machine with that name as an IP address. Because of this, use the **gethostbyname_r** subroutine.

Error Codes

The **gethostbyaddr_r** subroutine is unsuccessful if any of the following errors occur:

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	Indicates an unrecoverable error occured.
NO_ADDRESS	The requested *Name* parameter is valid but does not have an Internet address at the name server.
SERVICE_UNAVAILABLE	None of the name services specified are running or available.
EINVAL	The hostent pointer is NULL

Files

Item	Description
/etc/hosts	Contains the host name data base.
/etc/resolv.conf	Contains the name server and domain name.
/etc/netsvc.conf	Contains the name services ordering.
/usr/include/netdb.h	Contains the network database structure.

gethostbyname Subroutine

Purpose

Gets network host entry by name.

Library

```
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>

struct hostent *gethostbyname ( Name)
char *Name;
```

Description

The **gethostbyname** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **gethostbyname** subroutine retrieves host address and name information using a host name as a search key. Unless specified, the **gethostbyname** subroutine uses the default name services ordering, that is, it queries DNS/BIND, NIS or the local **/etc/hosts** file for the name.

When using DNS/BIND name service resolution, if the **/etc/resolv.conf** file exists, the **gethostbyname** subroutine queries the domain name server. The **gethostbyname** subroutine recognizes domain name servers as described in RFC883.

When using NIS for name resolution, if the **getdomaninname** subroutine is successful and **yp_bind** indicates yellow pages are running, then the **gethostbyname** subroutine queries NIS for the name.

The **gethostbyname** subroutine also searches the local **/etc/hosts** file for the name when indicated to do so.

The **gethostbyname** subroutine returns a pointer to a **hostent** structure, which contains information obtained from a name resolution services. The **hostent** structure is defined in the **netdb.h** header file.

Parameters

Item	Description
Name	Points to the host name.

Return Values

The **gethostbyname** subroutine returns a pointer to a **hostent** structure on success.

If the parameter *Name* passed to **gethostbyname** is actually an IP address, **gethostbyname** will return a non-NULL hostent structure with an IP address as the hostname without actually doing a lookup. Remember to call **inet_addr** subroutine to make sure *Name* is not an IP address before calling **gethostbyname**. To resolve an IP address call **gethostbyaddr** instead.

If an error occurs or if the end of the file is reached, the **gethostbyname** subroutine returns a null pointer and sets **h_errno** to indicate the error.

The environment variable, *NSORDER* can be set to overide the default name services ordering and the order specified in the **/etc/netsvc.conf** file.

By default, resolver routines first attempt to resolve names through the DNS/BIND, then NIS and the **/etc/hosts** file. The **/etc/netsvc.conf** file may specify a different search order. The environment variable NSORDER overrides both the **/etc/netsvc.conf** file and the default ordering. Services are ordered as **hosts =** *value*, *value*, *value* in the **/etc/netsvc.conf** file where at least one value must be specified from the list **bind**, **nis**, **local**. NSORDER specifies a list of values.

Error Codes

The **gethostbyname** subroutine is unsuccessful if any of the following errors occur:

Error	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.
SERVICE_UNAVAILABLE	None of the name services specified are running or available.

Examples

The following program fragment illustrates the use of the **gethostbyname** subroutine to look up a destination host:

```
hp=gethostbyname(argv[1]);
if(hp = = NULL) {
          fprintf(stderr, "rlogin: %s: unknown host\n", argv[1]);
          exit(2);
}
```

Files

Item	Description
/etc/hosts	Contains the host name data base.
/etc/resolv.conf	Contains the name server and domain name.
/etc/netsvc.conf	Contains the name services ordering.
/usr/include/netdb.h	Contains the network database structure.

Related reference
gethostbyaddr Subroutine
inet_addr Subroutine

Related information
Sockets Overview

gethostbyname_r Subroutine

Purpose

Gets network host entry by name.

Library

```
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)
```

Syntax

```
#include netdb.h>
int gethostbyname_r(Name, Htent, Ht_data)

const char *Name, struct hostent *Htent, struct hostent_data *Ht_data;
```

Description

This function internally calls the **gethostbyname** subroutine and stores the value returned by the **gethostbyname** subroutine to the hostent structure.

Parameters

Item	Description
Name	Points to the host name (which is a constant).
Htent	Points to a **hostent** structure in which the return value of the **gethostbyname** subroutine is stored.
Ht_data	Points to a **hostent_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Note:

The return value of the **gethostbyname** subroutine points to static data that is overwritten by subsequent calls. This data must be copied at every call to be saved for use by subsequent calls. The **gethostbyname_r** subroutine solves this problem.

If the *Name* parameter is an IP address, this subroutine searches for a machine with that IP address as a name. Because of this, use the **gethostbyaddr_r** subroutine instead of the **gethostbyname_r** subroutine if the *Name* parameter is an IP address.

Error Codes

The **gethostbyname_r** subroutine is unsuccessful if any of the following errors occurs:

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	An unrecoverable error occurred.
NO_ADDRESS	The requested *Name* is valid but does not have an Internet address at the name server.
SERVICE_UNAVAILABLE	None of the name services specified are running or available.
EINVAL	The hostent pointer is NULL.

Files

Item	Description
/etc/hosts	Contains the host name data base.
/etc/resolv.conf	Contains the name server and domain name.
/etc/netsvc.conf	Contains the name services ordering.
/usr/include/netdb.h	Contains the network database structure.

gethostent Subroutine

Purpose

Retrieves a network host entry.

Library

```
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>

struct hostent *gethostent ()
```

Description

The **gethostent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

When using DNS/BIND name service resolution, the **gethostent** subroutine is not defined.

When using NIS name service resolution or searching the local **/etc/hosts** file, the **gethostent** subroutine reads the next line of the **/etc/hosts** file, opening the file if necessary.

The **gethostent** subroutine returns a pointer to a **hostent** structure, which contains the equivalent fields for a host description line in the **/etc/hosts** file. The **hostent** structure is defined in the **netdb.h** file.

Return Values

Upon successful completion, the **gethostent** subroutine returns a pointer to a **hostent** structure.

If an error occurs or the end of the file is reached, the **gethostent** subroutine returns a null pointer.

Files

Item	Description
/etc/hosts	Contains the host name database.
/etc/netsvc.conf	Contains the name services ordering.
/usr/include/netdb.h	Contains the network database structure.

Related information
Sockets Overview
Network Address Translation

gethostent_r Subroutine

Purpose

Retrieves a network host entry.

Library

```
Standard C Library (libc.a)
(libbind)
(libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>

int gethostent_r (htent, ht_data)
struct hostent *htent;
struct hostent_data *ht_data;
```

Description

When using DNS/BIND name service resolution, the **gethostent_r** subroutine is not defined.

When using NIS name service resolution or searching the local **/etc/hosts** file, the **gethostent_r** subroutine reads the next line of the **/etc/hosts** file, and opens the file if necessary.

The **gethostent_r** subroutine internally calls the **gethostent** subroutine, and stores the values in the htent and ht_data structures.

The **gethostent** subroutine overwrites the static data returned in subsequent calls. The **gethostent_r** subroutine does not.

Parameters

Item	Description
htent	Points to the **hostent** structure
ht_data	Points to the **hostent_data** structure

Return Values

This subroutine returns a 0 if successful, and a -1 if unsuccessful.

Files

Item	Description
/etc/hosts	Contains the host name database.
/etc/netsvc.conf	Contains the name services ordering.
/usr/include/netdb.h	Contains the network database structure.

gethostid Subroutine

Purpose

Gets the unique identifier of the current host.

Library

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

Syntax

```
#include <unistd.h>

int gethostid ( )
```

Description

The **gethostid** subroutine allows a process to retrieve the 32-bit identifier for the current host. In most cases, the host ID is stored in network standard byte order and is a DARPA Internet Protocol address for a local machine.

All applications containing the **gethostid** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Return Values

Upon successful completion, the **gethostid** subroutine returns the identifier for the current host.

Related reference
sethostname Subroutine
Related information
Sockets Overview

gethostname Subroutine

Purpose

Gets the name of the local host.

Library

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

Syntax

```
#include <unistd.h>

int gethostname ( Name,  NameLength)
char *Name;
size_t NameLength;
```

Description

The **gethostname** subroutine retrieves the standard host name of the local host. If excess space is provided, the returned *Name* parameter is null-terminated. If insufficient space is provided, the returned name is truncated to fit in the given space. System host names are limited to 256 characters.

The **gethostname** subroutine allows a calling process to determine the internal host name for a machine on a network.

All applications containing the **gethostname** subroutine must be compiled with the **_BSD** macro 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
Name	Specifies the address of an array of bytes where the host name is to be stored.
NameLength	Specifies the length of the *Name* array.

Return Values

Upon successful completion, the system returns a value of 0.

If the **gethostname** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **gethostname** subroutine is unsuccessful if the following is true:

Error	Description
EFAULT	The *Name* parameter or *NameLength* parameter gives an invalid address.

Related reference
sethostname Subroutine
Related information
Sockets Overview

GetMultipleCompletionStatus Subroutine

Purpose

Dequeues multiple completion packets from a specified I/O completion port.

Syntax

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

```
int GetMultipleCompletionStatus (CompletionPort, Nmin, Nmax, Timeout, Results[])
HANDLE CompletionPort;
DWORD Nmin, Nmax, Timeout;
struct gmcs {
    DWORD transfer_count, completion_key, errorno;
    LPOVERLAPPED overlapped;
} Results[];
```

Description

The **GetMultipleCompletionStatus** subroutine attempts to dequeue a number of completion packets from the completion port that is specified by the *CompletionPort* parameter. The number of dequeued completion packets that are wanted ranges from the value of the *Nmin* parameter through the value of the *Nmax* parameter. As it collects the packets, this subroutine might wait a predetermined maximum amount of time that is specified by the *Timeout* parameter for the minimum number of completion packets to arrive. If, for example, the X^{th} completion packet does not arrive in time, the subroutine returns with only X-1 packets completed.

Either the *Timeout* parameter or a signal might cause a return with completions fewer than the value of the *Nmin* parameter. In other words, *Nmin* completions are not guaranteed to be returned unless the *Timeout* parameter value is set to INFINITE, and a signal does not interrupt the wait. The return of zero completions is not considered an error. The **errno** value will, however, indicate the condition with either the **ETIMEDOUT** or **EINTR** error code. In extreme low-memory situations, the kernel might not be able to provide a timeout. In this case, the system call returns immediately with any available completions, up to the value of the *Nmax* parameter, and the **errno** value is set to **ENOMEM**. Be sure to set the **errno** value to zero before calling the **GetMultipleCompletionStatus** subroutine so that the change of the **errno** value that the subroutine makes can be distinguished from the existing value.

The **GetMultipleCompletionStatus** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note:

1. This subroutine only works with file descriptors of sockets, or regular files for use with the asynchronous I/O subsystem. It does not work with file descriptors of other types.
2. This function must be the exclusive wait mechanism on a completion port. Multiple simultaneous waits through the **GetMultipleCompletionStatus** subroutine, the **GetQueuedCompletionStatus** subroutine, or both, are not supported.
3. When the **GetMultipleCompletionStatus** subroutine is used with the **lio_listio** subroutine, you can set the value of the *cmd* parameter of the **lio_listio** subroutine to **LIO_NOWAIT_GMCS** to avoid asynchronous updating of the **aiocb** structures, thereby reducing overhead. In this case, you must use the **GetMultipleCompletionStatus** subroutine to wait for I/O completions, and retrieve completion status only from the **struct gmcs** members, not from the **aiocb** structure. When using the **LIO_NOWAIT_GMCS** value, do not use the *completion_key* value in the **gmcs** structure. Do not use the **LIO_NOWAIT_AIOWAIT** value with the **lio_listio** subroutine when using the **GetMultipleCompletionStatus** subroutine. The **LIO_NOWAIT_GMCS** value is available for that purpose.
4. Cancelling an asynchronous I/O operation will not affect the **GetMultipleCompletionStatus** subroutine. Even if the cancelling reduces the number of active asynchronous I/O operations to zero, the subroutine will continue to wait.
5. When using the **GetMultipleCompletionStatus** subroutine with sockets, do not wait for multiple completions (*Nmin* > 1) with an **INFINITE** timeout. Use a finite timeout value, and to be prepared to repeat the call if additional completions are still expected.

Parameters

Item	Description	Attribute description
CompletionPort	Specifies the file descriptor for the completion port that this subroutine will access.	
Nmin	Specifies the minimum number of completions. Fewer might be returned if the value of the *timeout* parameter is exceeded, or a signal accepted. More might be returned, up to the number that is specified by the *Nmax* parameter, if additional completions have occurred. Setting the value of the *Nmin* parameter to zero will poll for completions and return immediately, ignoring the value of the *timeout* parameter.	
Nmax	Specifies the maximum number of completions to wait for, up to the value of the GMCS_NMAX macro.	
Results	This is the address of an array of the **gmcs** structure to receive the completion data. The array must contain space for the number of entries specified by the *Nmax* parameter.	
	Results[i]. transfer_count	Specifies the number of bytes transferred. This parameter is set by the subroutine from the value received in the i^{th} completion packet. This value is limited to 2 G.
	Results[i].completion_key	Specifies the completion key associated with the file descriptor that is used in the transfer request. This parameter is set by the subroutine from the value received in the i^{th} completion packet. Do not use this value with the **LIO_NOWAIT_GMCS** command parameter of the **lio_listio** subroutine.
	Results[i].errorno	Specifies the **errno** value that is associated with the i^{th} completion packet. When asynchronous I/O requests are started using the **lio_listio** subroutine with the **LIO_NOWAIT_GMCS** command parameter, you must use this error value, not the **aio_errno** member in the **aiocb** structure, to retrieve the error value that is associated with an I/O request.

Item	Description	Attribute description
	Results[i].overlapped	Specifies the overlapped structure that is used in the transfer request. This parameter is set by the subroutine from the value received in the i^{th} completion packet. For regular files, this parameter contains a pointer to the asynchronous I/O control block (AIOCB) for a completed AIO request. If an application uses the same completion port for both socket and AIO to regular files, it must use unique *completion_key* values to differentiate between sockets and regular files to properly interpret the *overlapped* parameter.
Timeout	Specifies the amount of time in milliseconds that the subroutine is to wait for completion packets. This value can be set to zero. If this parameter is set to INFINITE, the subroutine will never time out.	

Return Values

Item	Description
Success	The subroutine returns an integer ranging from zero through the value of the *Nmax* parameter, indicating how many completion packets are dequeued.
Failure	The subroutine returns a value of -1.

Error codes

The subroutine is unsuccessful if any of the following errors occur:

Item	Description
EINVAL	The value of the *CompletionPort* or other parameter is not valid.
EBUSY	Another thread is already waiting on the I/O completion port.
EBADF	This error code might also be returned when the value of the *CompletionPort* parameter is not valid.

If an error occurs after some completions have been handled, the error notifications will be lost. An **EFAULT** error when copying out results can cause the situation.

Examples

1. The following program fragment illustrates the use of the **GetMultipleCompletionStatus** subroutine to dequeue up to 10 completion packets within a 100-millisecond window.

```
struct gmcs results[10];
int n_results;
HANDLE iocpfd;
errno = 0;
n_results = GetMultipleCompletionStatus(iocpfd, 10, 10, 100, results);
```

Related information

lio_listio subroutine

Error Notification Object Class

getnameinfo Subroutine

Purpose

Address-to-host name translation [given the binary address and port].

Note: This is the reverse functionality of the “getaddrinfo Subroutine” on page 144 host-to-address translation.

Attention: This is not a POSIX (1003.1g) specified function.

Library

Library (**libc.a**)

Syntax

```
#include <sys/socket.h>
#include <netdb.h>
int
getnameinfo (sa, salen, host, hostlen, serv, servlen, flags)
const struct sockaddr *sa;
char *host;
size_t hostlen;
char *serv;
size_t servlen;
int flags;
```

Description

The *sa* parameter points to either a **sockaddr_in** structure (for IPv4) or a **sockaddr_in6** structure (for IPv6) that holds the IP address and port number. The*salen* parameter gives the length of the **sockaddr_in** or **sockaddr_in6** structure.

Note: A reverse lookup is performed on the IP address and port number provided in *sa*.

The *host* parameter is a buffer where the hostname associated with the IP address is copied. The *hostlen* parameter provides the length of this buffer. The service name associated with the port number is copied into the buffer pointed to by the *serv* parameter. The *servlen* parameter provides the length of this buffer.

The *flags* parameter defines flags that may be used to modify the default actions of this function. By default, the fully-qualified domain name (FQDN) for the host is looked up in DNS and returned.

Item	Description
NI_NOFQDN	If set, return only the hostname portion of the FQDN. If cleared, return the FQDN.
NI_NUMERICHOST	If set, return the numeric form of the host address. If cleared, return the name.
NI_NAMEREQD	If set, return an error if the host's name cannot be determined. If cleared, return the numeric form of the host's address (as if NI_NUMERICHOST had been set).
NI_NUMERICSERV	If set, return the numeric form of the desired service. If cleared, return the service name.
NI_DGRAM	If set, consider the desired service to be a datagram service, (for example, call **getservbyport** with an argument of **udp**). If clear, consider the desired service to be a stream service (for example, call **getserbyport** with an argument of **tcp**).

Return Values

A zero return value indicates successful completion; a non-zero value indicates failure. If successful, the strings for hostname and service name are copied into the *host* and *serv* buffers, respectively. If either the host or service name cannot be located, the numeric form is copied into the *host* and *serv* buffers, respectively.

Related information
gai_strerror Subroutine
Subroutines Overview

getnetbyaddr Subroutine

Purpose

Gets network entry by address.

Library

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

Syntax

```
#include <netdb.h>

struct netent *getnetbyaddr (Network, Type)
long Network;
int Type;
```

Description

The **getnetbyaddr** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getnetbyaddr** subroutine retrieves information from the **/etc/networks** file using the network address as a search key. The **getnetbyaddr** subroutine searches the file sequentially from the start of the file until it encounters a matching net number and type or until it reaches the end of the file.

The **getnetbyaddr** subroutine returns a pointer to a **netent** structure, which contains the equivalent fields for a network description line in the **/etc/networks** file. The **netent** structure is defined in the **netdb.h** file.

Use the **endnetent** subroutine to close the **/etc/networks** file.

All applications containing the **getnetbyaddr** subroutine must be compiled with the **_BSD** macro 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
Network	Specifies the number of the network to be located.
Type	Specifies the address family for the network. The only supported value is **AF_INET**.

Return Values

Upon successful completion, the **getnetbyaddr** subroutine returns a pointer to a **netent** structure.

If an error occurs or the end of the file is reached, the **getnetbyaddr** subroutine returns a null pointer.

Files

Item	Description
/etc/networks	Contains official network names.

Related reference
endnetent Subroutine
setnetent Subroutine
Related information
Sockets Overview

getnetbyaddr_r Subroutine

Purpose

Gets network entry by address.

Library

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

Syntax

```
#include<netdb.h>
int getnetbyaddr_r(net, type, netent, net_data)

register in_addr_t net;
register int type;
struct netent *netent;
struct netent_data *net_data;
```

Description

The **getnetbyaddr_r** subroutine retrieves information from the **/etc/networks** file using the *Name* parameter as a search key.

The **getnetbyaddr_r** subroutine internally calls the **getnetbyaddr** subroutine and stores the information in the structure data.

The **getnetbyaddr** subroutine overwrites the static data returned in subsequent calls. The **getnetbyaddr_r** subroutine does not.

Use the **endnetent_r** subroutine to close the **/etc/networks** file.

Parameters

Item	Description
Net	Specifies the number of the network to be located.
Type	Specifies the address family for the network. The only supported values are AF_INET, and AF_INET6.
netent	Points to the **netent** structure.
net_data	Points to the **net_data structure**.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Files

Item	Description
/etc/networks	Contains official network names.

getnetbyname Subroutine

Purpose

Gets network entry by name.

Library

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

Syntax

```
#include <netdb.h>

struct netent *getnetbyname (Name)
char *Name;
```

Description

The **getnetbyname** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getnetbyname** subroutine retrieves information from the **/etc/networks** file using the *Name* parameter as a search key. The **getnetbyname** subroutine searches the **/etc/networks** file sequentially from the start of the file until it encounters a matching net name or until it reaches the end of the file.

The **getnetbyname** subroutine returns a pointer to a **netent** structure, which contains the equivalent fields for a network description line in the **/etc/networks** file. The **netent** structure is defined in the **netdb.h** file.

Use the **endnetent** subroutine to close the **/etc/networks** file.

All applications containing the **getnetbyname** subroutine must be compiled with the **_BSD** macro 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
Name	Points to a string containing the name of the network.

Return Values

Upon successful completion, the **getnetbyname** subroutine returns a pointer to a **netent** structure.

If an error occurs or the end of the file is reached, the **getnetbyname** subroutine returns a null pointer.

Files

Item	Description
/etc/networks	Contains official network names.

Related reference
endnetent Subroutine
Related information
Sockets Overview

getnetbyname_r Subroutine

Purpose

Gets network entry by name.

Library

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

Syntax

```
#include <netdb.h>
int getnetbyname_r(Name, netent, net_data)
register const char *Name;
struct netent *netent;
struct netent_data *net_data;
```

Description

The **getnetbyname_r** subroutine retrieves information from the **/etc/networks** file using the *Name* parameter as a search key.

The **getnetbyname_r** subroutine internally calls the **getnetbyname** subroutine and stores the information in the structure data.

The **getnetbyname** subroutine overwrites the static data returned in subsequent calls. The **getnetbyname_r** subroutine does not.

Use the **endnetent_r** subroutine to close the **/etc/networks** file.

Parameters

Item	Description
Name	Points to a string containing the name of the network.
netent	Points to the **netent** structure.
net_data	Points to the **net_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the **getnetbyname_r** subroutine returns a -1 to indicate error.

Files

Item	Description
/etc/networks	Contains official network names.

getnetent Subroutine

Purpose

Gets network entry.

Library

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

Syntax

```
#include <netdb.h>
```

```
struct netent *getnetent ( )
```

Description

The **getnetent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getnetent** subroutine retrieves network information by opening and sequentially reading the **/etc/networks** file.

The **getnetent** subroutine returns a pointer to a **netent** structure, which contains the equivalent fields for a network description line in the **/etc/networks** file. The **netent** structure is defined in the **netdb.h** file.

Use the **endnetent** subroutine to close the **/etc/networks** file.

All applications containing the **getnetent** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Return Values

Upon successful completion, the **getnetent** subroutine returns a pointer to a **netent** structure.

If an error occurs or the end of the file is reached, the **getnetent** subroutine returns a null pointer.

Files

Item	Description
/etc/networks	Contains official network names.

Related reference
setnetent Subroutine
Related information
Sockets Overview

getnetent_r Subroutine

Purpose

Gets network entry.

Library

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

Syntax

```
#include <netdb.h>
int getnetent_r(netent, net_data)

struct netent *netent;
struct netent_data *net_data;
```

Description

The **getnetent_r** subroutine retrieves network information by opening and sequentially reading the **/etc/networks** file. This subroutine internally calls the **getnetent** subroutine and stores the values in the hostent structure.

The **getnetent** subroutine overwrites the static data returned in subsequent calls. The **getnetent_r** subroutine does not. Use the **endnetent_r** subroutine to close the **/etc/networks** file.

Parameters

Item	Description
netent	Points to the **netent** structure.
net_data	Points to the **net_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the **getnetent_r** subroutine returns a -1 to indicate error.

Files

Item	Description
/etc/networks	Contains official network names.

getnetgrent_r Subroutine

Purpose

Handles the group network entries.

Library

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

Syntax

```
#include<netdb.h>
int getnetgrent_r(machinep, namep, domainp, ptr)
 char **machinep, **namep, **domainp;
void **ptr;
```

Description

The **getnetgrent_r** subroutine internally calls the **getnetgrent** subroutine and stores the information in the structure data. This subroutine returns 1 or 0, depending if netgroup contains the machine, user, and domain triple as a member. Any of these three strings can be NULL, in which case it signifies a wildcard.

The **getnetgrent_r** subroutine returns the next member of a network group. After the call, the *machinep* parameter contains a pointer to a string containing the name of the machine part of the network group member. The *namep* and *domainp* parameters contain similar pointers. If *machinep*, *namep*, or *domainp* is returned as a NULL pointer, it signifies a wildcard.

The **getnetgrent** subroutine overwrites the static data returned in subsequent calls. The **getnetgrent_r** subroutine does not.

Parameters

Item	Description
machinep	Points to the string containing the machine part of the network group.
namep	Points to the string containing the user part of the network group.
domainp	Points to the string containing the domain name.
ptr	Keeps the function threadsafe.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Files

Item	Description
/etc/netgroup	Contains network groups recognized by the system.
/usr/include/netdb.h	Contains the network database structures.

getpeername Subroutine

Purpose

Gets the name of the peer socket.

Library

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

Syntax

```
#include <sys/socket.h>
int getpeername ( Socket,  Name,  NameLength)
```

```
int Socket;
struct sockaddr *Name;
socklen_t *NameLength;
```

Description

The **getpeername** subroutine retrieves the *Name* parameter from the peer socket connected to the specified socket. The *Name* parameter contains the address of the peer socket upon successful completion.

A process created by another process can inherit open sockets. The created process may need to identify the addresses of the sockets it has inherited. The **getpeername** subroutine allows a process to retrieve the address of the peer socket at the remote end of the socket connection.

Note: The **getpeername** subroutine operates only on connected sockets.

A process can use the **getsockname** subroutine to retrieve the local address of a socket.

All applications containing the **getpeername** subroutine must be compiled with the **_BSD** macro 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
Socket	Specifies the descriptor number of a connected socket.
Name	Points to a **sockaddr** structure that contains the address of the destination socket upon successful completion. The **/usr/include/sys/socket.h** file defines the **sockaddr** structure.
NameLength	Points to the size of the address structure. Initializes the *NameLength* parameter to indicate the amount of space pointed to by the *Name* parameter. Upon successful completion, it returns the actual size of the *Name* parameter returned.

Return Values

Upon successful completion, a value of 0 is returned and the *Name* parameter holds the address of the peer socket.

If the **getpeername** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **getpeername** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
EINVAL	The socket has been shut down.
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.

Examples

The following program fragment illustrates the use of the **getpeername** subroutine to return the address of the peer connected on the other end of the socket:

```
struct sockaddr_in name;
int namelen = sizeof(name);
.
.
.
if(getpeername(0,(struct sockaddr*)&name, &namelen)<0){
  syslog(LOG_ERR,"getpeername: %m");
  exit(1);
} else
  syslog(LOG_INFO,"Connection from %s",inet_ntoa(name.sin_addr));
.
.
.
```

Related reference

getsockname Subroutine

Related information

Sockets Overview

getprotobyname Subroutine

Purpose

Gets protocol entry from the **/etc/protocols** file by protocol name.

Library

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

Syntax

```
#include <netdb.h>
```

struct protoent *getprotobyname (*Name***)**
char ***Name;**

Description

The **getprotobyname** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getprotobyname** subroutine retrieves protocol information from the **/etc/protocols** file by protocol name. An application program can use the **getprotobyname** subroutine to access a protocol name, its aliases, and protocol number.

The **getprotobyname** subroutine searches the **protocols** file sequentially from the start of the file until it finds a matching protocol name or until it reaches the end of the file. The subroutine returns a pointer to a **protoent** structure, which contains fields for a line of information in the **/etc/protocols** file. The **netdb.h** file defines the **protoent** structure.

Use the **endprotoent** subroutine to close the **/etc/protocols** file.

All applications containing the **getprotobyname** subroutine must be compiled with the **_BSD** macro 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
Name	Specifies the protocol name.

Return Values

Upon successful completion, the **getprotobyname** subroutine returns a pointer to a **protoent** structure.

If an error occurs or the end of the file is reached, the **getprotbyname** subroutine returns a null pointer.

Related reference
endprotoent Subroutine
setprotoent Subroutine
setservent Subroutine
Related information
Sockets Overview

getprotobyname_r Subroutine

Purpose

Gets protocol entry from the **/etc/protocols** file by protocol name.

Library

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

Syntax

```
#include <netdb.h>

int  getprotobyname_r(Name, protoent, proto_data)
register const char *Name;
struct protoent *protoent;
struct protoent_data *proto_data;
```

Description

The **getprotobyname_r** subroutine retrieves protocol information from the **/etc/protocols** file by protocol name.

An application program can use the **getprotobyname_r** subroutine to access a protocol name, aliases, and protocol number.

The **getprotobyname_r** subroutine searches the protocols file sequentially from the start of the file until it finds a matching protocol name or until it reaches the end of the file. The subroutine writes the protoent structure, which contains fields for a line of information in the **/etc/protocols** file.

The **netdb.h** file defines the protoent structure.

The **getprotobyname** subroutine overwrites any static data returned in subsequent calls. The **getprotobyname_r** subroutine does not.

Use the **endprotoent_r** subroutine to close the **/etc/protocols** file.

Parameters

Item	Description
Name	Specifies the protocol name.
protoent	Points to the **protoent** structure.
proto_data	Points to the **proto_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the **getprotobyname_r** subroutine returns a -1 to indicate error.

getprotobynumber Subroutine

Purpose

Gets a protocol entry from the **/etc/protocols** file by number.

Library

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

Syntax

```
#include <netdb.h>

struct protoent *getprotobynumber ( Protocol)
int Protocol;
```

Description

The **getprotobynumber** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getprotobynumber** subroutine retrieves protocol information from the **/etc/protocols** file using a specified protocol number as a search key. An application program can use the **getprotobynumber** subroutine to access a protocol name, its aliases, and protocol number.

The **getprotobynumber** subroutine searches the **/etc/protocols** file sequentially from the start of the file until it finds a matching protocol name or protocol number, or until it reaches the end of the file. The subroutine returns a pointer to a **protoent** structure, which contains fields for a line of information in the **/etc/protocols** file. The **netdb.h** file defines the **protoent** structure.

Use the **endprotoent** subroutine to close the **/etc/protocols** file.

All applications containing the **getprotobynumber** subroutine must be compiled with the **_BSD** macro 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
Protocol	Specifies the protocol number.

Return Values

Upon successful completion, the **getprotobynumber** subroutine, returns a pointer to a **protoent** structure.

If an error occurs or the end of the file is reached, the **getprotobynumber** subroutine returns a null pointer.

Files

Item	Description
/etc/protocols	Contains protocol information.

Related reference
endprotoent Subroutine
Related information
Sockets Overview

getprotobynumber_r Subroutine

Purpose

Gets a protocol entry from the **/etc/protocols** file by number.

Library

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

Syntax

```
#include <netdb.h>
int getprotobynumber_r(proto, protoent, proto_data)
register int proto;
struct protoent *protoent;
struct protoent_data *proto_data;
```

Description

The **getprotobynumber_r** subroutine retrieves protocol information from the **/etc/protocols** file using a specified protocol number as a search key.

An application program can use the **getprotobynumber_r** subroutine to access a protocol name, aliases, and number.

The **getprotobynumber_r** subroutine searches the **/etc/protocols** file sequentially from the start of the file until it finds a matching protocol name, protocol number, or until it reaches the end of the file.

The subroutine writes the protoent structure, which contains fields for a line of information in the **/etc/protocols** file.

The **netdb.h** file defines the protoent structure.

The **getprotobynumber** subroutine overwrites static data returned in subsequent calls. The **getprotobynumber_r** subroutine does not.

Use the **endprotoent_r** subroutine to close the **/etc/protocols** file.

Parameters

Item	Description
proto	Specifies the protocol number.
protoent	Points to the **protoent** structure.
proto_data	Points to the **proto_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the **getprotobynumber_r** subroutine sets the *protoent* parameter to NULL and returns a -1 to indicate error.

Files

Item	Description
/etc/protocols	Contains protocol information.

getprotoent Subroutine

Purpose

Gets protocol entry from the **/etc/protocols** file.

Library

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

Syntax

```
#include <netdb.h>
```

```
struct protoent *getprotoent ( )
```

Description

The **getprotoent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getprotoent** subroutine retrieves protocol information from the **/etc/protocols** file. An application program can use the **getprotoent** subroutine to access a protocol name, its aliases, and protocol number.

The **getprotoent** subroutine opens and performs a sequential read of the **/etc/protocols** file. The **getprotoent** subroutine returns a pointer to a **protoent** structure, which contains the fields for a line of information in the **/etc/protocols** file. The **netdb.h** file defines the **protoent** structure.

Use the **endprotoent** subroutine to close the **/etc/protocols** file.

All applications containing the **getprotoent** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Return Values

Upon successful completion, the **getprotoent** subroutine returns a pointer to a **protoent** structure.

If an error occurs or the end of the file is reached, the **getprotoent** subroutine returns a null pointer.

Files

Item	Description
/etc/protocols	Contains protocol information.

Related reference
endprotoent Subroutine
Related information
Sockets Overview

getprotoent_r Subroutine

Purpose

Gets protocol entry from the **/etc/protocols** file.

Library

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

Syntax

```
#include <netdb.h>

int getprotoent_r(protoent, proto_data)
struct protoent *protoent;
struct protoent_data *proto_data;
```

Description

The **getprotoent_r** subroutine retrieves protocol information from the **/etc/protocols** file. An application program can use the **getprotoent_r** subroutine to access a protocol name, its aliases, and protocol number. The **getprotoent_r** subroutine opens and performs a sequential read of the **/etc/protocols** file. This subroutine writes to the protoent structure, which contains the fields for a line of information in the **/etc/protocols** file.

The **netdb.h** file defines the protoent structure.

Use the **endprotoent_r** subroutine to close the **/etc/protocols** file. Static data is overwritten in subsequent calls when using the **getprotoent** subroutine. The **getprotoent_r** subroutine does not overwrite.

Parameters

Item	Description
protoent	Points to the **protoent** structure.
proto_data	Points to the **proto_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the **getprotoent_r** subroutine sets the *protoent* parameter to NULL.

Files

Item	Description
/etc/protocols	Contains protocol information.

GetQueuedCompletionStatus Subroutine

Purpose

Dequeues a completion packet from a specified I/O completion port.

Syntax

```
#include <iocp.h>
boolean_t GetQueuedCompletionStatus (CompletionPort, TransferCount, CompletionKey, Overlapped,
Timeout)
HANDLE CompletionPort;
LPDWORD TransferCount, CompletionKey;
LPOVERLAPPED Overlapped; DWORD Timeout;
```

Description

The **GetQueuedCompletionStatus** subroutine attempts to dequeue a completion packet from the *CompletionPort* parameter. If there is no completion packet to be dequeued, this subroutine waits a predetermined amount of time as indicated by the *Timeout* parameter for a completion packet to arrive.

The **GetQueuedCompletionStatus** subroutine returns a boolean indicating whether or not a completion packet has been dequeued.

The **GetQueuedCompletionStatus** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note: This subroutine only works with file descriptors of sockets, or regular files for use with the Asynchronous I/O (AIO) subsystem. It does not work with file descriptors of other types.

Parameters

Item	Description
CompletionPort	Specifies the completion port that this subroutine will attempt to access.
TransferCount	Specifies the number of bytes transferred. This parameter is set by the subroutine from the value received in the completion packet.
CompletionKey	Specifies the completion key associated with the file descriptor used in the transfer request. This parameter is set by the subroutine from the value received in the completion packet.
Overlapped	Specifies the overlapped structure used in the transfer request. This parameter is set by the subroutine from the value received in the completion packet. For regular files, this parameter contains a pointer to the AIOCB for a completed AIO request. If an application uses the same completion port for both socket and AIO to regular files, it must use unique *CompletionKey* values to differentiate between sockets and regular files in order to properly interpret the *Overlapped* parameter.
Timeout	Specifies the amount of time in milliseconds the subroutine is to wait for a completion packet. If this parameter is set to INFINITE, the subroutine will never timeout.

Return Values

Upon successful completion, the **GetQueuedCompletionStatus** subroutine returns a boolean indicating its success.

If the **GetQueuedCompletionStatus** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

Error Codes

The subroutine is unsuccessful if any of the following errors occur:

Item	Description
ETIMEDOUT	No completion packet arrived to be dequeued and the *Timeout* parameter has elapsed.
EINVAL	The value of the *CompletionPort* or other parameter is not valid.
EAGAIN	Resource temporarily unavailable. If a sleep is interrupted by a signal, **EAGAIN** may be returned.
ENOTCONN	Socket is not connected. The **ENOTCONN** return can happen for two reasons. One is if a request is made, the **fd** is then closed, then the request is returned back to the process. The error will be **ENOTCONN**. The other is if the socket drops while the **fd** is still open, the requests after the socket drops (disconnects) will return **ENOTCONN**.
EBADF	This error code might also be returned when the value of the *CompletionPort* parameter is not valid.

Examples

The following program fragment illustrates the use of the **GetQueuedCompletionStatus** subroutine to dequeue a completion packet.

```
int transfer_count, completion_key
LPOVERLAPPED overlapped;
c = GetQueuedCompletionStatus (34, &transfer_count, &completion_key, &overlapped, 1000);
```

Related information

Error Notification Object Class

getservbyname Subroutine

Purpose

Gets service entry by name.

Library

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

Syntax

```
#include <netdb.h>
```

```
struct servent *getservbyname ( Name,  Protocol)
char *Name, *Protocol;
```

Description

The **getservbyname** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getservbyname** subroutine retrieves an entry from the **/etc/services** file using the service name as a search key.

An application program can use the **getservbyname** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The **getservbyname** subroutine searches the **/etc/services** file sequentially from the start of the file until it finds one of the following:

- Matching name and protocol number
- Matching name when the *Protocol* parameter is set to 0
- End of the file

Upon locating a matching name and protocol, the **getservbyname** subroutine returns a pointer to the **servent** structure, which contains fields for a line of information from the **/etc/services** file. The **netdb.h** file defines the **servent** structure and structure fields.

Use the **endservent** subroutine to close the **/etc/services** file.

All applications containing the **getservbyname** subroutine must be compiled with the **_BSD** macro 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
Name	Specifies the name of a service.
Protocol	Specifies a protocol for use with the specified service.

Return Values

The **getservbyname** subroutine returns a pointer to a **servent** structure when a successful match occurs. Entries in this structure are in network byte order.

If an error occurs or the end of the file is reached, the **getservbyname** subroutine returns a null pointer.

Files

Item	Description
/etc/services	Contains service names.

Related reference
endservent Subroutine
Related information
Sockets Overview
Understanding Network Address Translation

getservbyname_r Subroutine

Purpose

Gets service entry by name.

Library

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

Syntax

```
#include <netdb.h>
int getservbyname_r(name, proto, servent, serv_data)
const char *Name, proto;
struct servent *servent;
struct servent_data *serv_data;
```

Description

Requirement: Use the **getservbyname** subroutine instead of the **getservbyname_r** subroutine. The **getservbyname_r** subroutine is compatible only with earlier versions of AIX.

An application program can use the **getservbyname_r** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The **getservbyname_r** subroutine searches the **/etc/services** file sequentially from the start of the file until it finds one of the following:

- Matching name and protocol number.
- Matching name when the *Protocol* parameter is set to 0.
- End of the file.

Upon locating a matching name and protocol, the **getservbyname_r** subroutine stores the values to the servent structure. The **getservbyname** subroutine overwrites the static data it returns in subsequent calls. The **getservbyname_r** subroutine does not.

Use the **endservent_r** subroutine to close the **/etc/hosts** file.

You must fill the **servent_data** structure with zeros before its first access by either the **setservent_r** or the **getservbyname_r** subroutine.

Parameters

Item	Description
name	Specifies the name of a service.
proto	Specifies a protocol for use with the specified service.
servent	Points to the **servent** structure.
serv_data	Points to the **serv_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful. The **getservbyname** subroutine returns a pointer to a servent structure when a successful match occurs. Entries in this structure are in network byte order.

Note: If an error occurs or the end of the file is reached, the **getservbyname_r** returns a -1.

Files

Item	Description
/etc/services	Contains service names.

getservbyport Subroutine

Purpose

Gets service entry by port.

Library

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

Syntax

```
#include <netdb.h>

struct servent *getservbyport (Port, Protocol)
int Port;char *Protocol;
```

Description

The **getservbyport** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getservbyport** subroutine retrieves an entry from the **/etc/services** file using a port number as a search key.

An application program can use the **getservbyport** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The **getservbyport** subroutine searches the services file sequentially from the beginning of the file until it finds one of the following:

- Matching protocol and port number
- Matching protocol when the *Port* parameter value equals 0
- End of the file

Upon locating a matching protocol and port number or upon locating a matching protocol only if the *Port* parameter value equals 0, the **getservbyport** subroutine returns a pointer to a **servent** structure, which contains fields for a line of information in the **/etc/services** file. The **netdb.h** file defines the **servent** structure and structure fields.

Use the **endservent** subroutine to close the **/etc/services** file.

All applications containing the **getservbyport** subroutine must be compiled with the **_BSD** macro 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
Port	Specifies the port where a service resides.
Protocol	Specifies a protocol for use with the service.

Return Values

Upon successful completion, the **getservbyport** subroutine returns a pointer to a **servent** structure.

If an error occurs or the end of the file is reached, the **getservbyport** subroutine returns a null pointer.

Files

Item	Description
/etc/services	Contains service names.

Related reference
endservent Subroutine
endprotoent Subroutine
Related information
Sockets Overview

getservbyport_r Subroutine

Purpose

Gets service entry by port.

Library

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

Syntax

```
#include <netdb.h>

int getservbyport_r(Port, Proto, servent, serv_data)
int Port;
const char *Proto;
struct servent *servent;
struct servent_data *serv_data;
```

Description

The **getservbyport_r** subroutine retrieves an entry from the **/etc/services** file using a port number as a search key. An application program can use the **getservbyport_r** subroutine to access a service, service aliases, the protocol for the service, and a protocol port number for the service.

The **getservbyport_r** subroutine searches the services file sequentially from the beginning of the file until it finds one of the following:

- Matching protocol and port number
- Matching protocol when the Port parameter value equals 0
- End of the file

Upon locating a matching protocol and port number or upon locating a matching protocol where the *Port* parameter value equals 0, the **getservbyport_r** subroutine returns a pointer to a servent structure, which contains fields for a line of information in the **/etc/services** file. The **netdb.h** file defines the servent structure, the servert_data structure, and their fields.

The **getservbyport** routine overwrites static data returned on subsequent calls. The **getservbyport_r** routine does not.

Use the **endservent_r** subroutine to close the **/etc/services** file.

Parameters

Item	Description
Port	Specifies the port where a service resides.
Proto	Specifies a protocol for use with the service.
servent	Points to the **servent** structure.
serv_data	Points to the **serv_data** structure.

Return Values

The function returns a 0 if successful and a -1 if unsuccessful.

Note: If an error occurs or the end of the file is reached, the **getservbyport_r** subroutine returns a -1 to indicate error.

Files

Item	Description
/etc/services	Contains service names.

getservent Subroutine

Purpose

Gets services file entry.

Library

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

Syntax

```
#include <netdb.h>
```

```
struct servent *getservent ( )
```

Description

The **getservent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **getservent** subroutine opens and reads the next line of the **/etc/services** file.

An application program can use the **getservent** subroutine to retrieve information about network services and the protocol ports they use.

The **getservent** subroutine returns a pointer to a **servent** structure, which contains fields for a line of information from the **/etc/services** file. The **servent** structure is defined in the **netdb.h** file.

The **/etc/services** file remains open after a call by the **getservent** subroutine. To close the **/etc/services** file after each call, use the **setservent** subroutine. Otherwise, use the **endservent** subroutine to close the **/etc/services** file.

All applications containing the **getservent** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Return Values

The **getservent** subroutine returns a pointer to a **servent** structure when a successful match occurs.

If an error occurs or the end of the file is reached, the **getservent** subroutine returns a null pointer.

Files

Item	Description
/etc/services	Contains service names.

Related reference
endprotoent Subroutine
Related information
Sockets Overview
Understanding Network Address Translation

getservent_r Subroutine

Purpose

Gets services file entry.

Library

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

Syntax

```
#include <netdb.h>
int getservent_r(servent, serv_data)
struct servent *servent;
struct servent_data *serv_data;
```

Description

The **getservent_r** subroutine opens and reads the next line of the **/etc/services** file.An application program can use the **getservent_r** subroutine to retrieve information about network services and the protocol ports they use.

The **/etc/services** file remains open after a call by the **getservent_r** subroutine. To close the **/etc/services** file after each call, use the **setservent_r** subroutine. Otherwise, use the **endservent_r** subroutine to close the **/etc/services** file.

Parameters

Item	Description
servent	Points to the **servent** structure.
serv_data	Points to the **serv_data** structure.

Return Values

The **getservent_r** fails when a successful match occurs. The**getservent** subroutine overwrites static data returned on subsequent calls. The **getservent_r** subroutine does not.

Files

Item	Description
/etc/services	Contains service names.

getsockname Subroutine

Purpose

Gets the socket name.

Library

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

Syntax

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

int getsockname (Socket, Name, NameLength)
int  Socket;
struct sockaddr * Name;
socklen_t * NameLength;
```

Description

The **getsockname** subroutine retrieves the locally bound address of the specified socket. The socket address represents a port number in the Internet domain and is stored in the **sockaddr** structure pointed to by the *Name* parameter. The **sys/socket.h** file defines the **sockaddr** data structure.

A process created by another process can inherit open sockets. To use the inherited socket, the created process needs to identify their addresses. The **getsockname** subroutine allows a process to retrieve the local address bound to the specified socket.

A process can use the **getpeername** subroutine to determine the address of a destination socket in a socket connection.

Parameters

Item	Description
Socket	Specifies the socket for which the local address is desired.
Name	Points to the structure containing the local address of the specified socket.
NameLength	Specifies the size of the local address in bytes. Initializes the value pointed to by the *NameLength* parameter to indicate the amount of space pointed to by the *Name* parameter.

Return Values

Upon successful completion, a value of 0 is returned, and the *NameLength* parameter points to the size of the socket address.

If the **getsockname** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

- For sockets in the AF_UNIX domain, if the returned value of the **NameLength** parameter is greater than 255, the corresponding value of the **sun_len** field in the overloaded sockaddr structure is assigned an address of 0xFF because of the bit size limitations of the **sun_len** field.

Error Codes

The **getsockname** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOBUFS	Insufficient resources are available in the system to complete the call.
EFAULT	The *Address* parameter is not in a writable part of the user address space.

Related reference
getpeername Subroutine
socket Subroutine
socks5tcp_bind Subroutine
socks5tcp_connect Subroutine
Related information
Checking for Pending Connections Example Program
Sockets Overview

getsockopt Subroutine

Purpose

Gets options on sockets.

Library

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

Syntax

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

int getsockopt (Socket, Level, OptionName, OptionValue, OptionLength)
int  Socket,  Level,  OptionName;
void * OptionValue;
socklen_t * OptionLength;
```

Description

The **getsockopt** subroutine allows an application program to query socket options. The calling program specifies the name of the socket, the name of the option, and a place to store the requested information. The operating system gets the socket option information from its internal data structures and passes the requested information back to the calling program.

Options can exist at multiple protocol levels. They are always present at the uppermost socket level. When retrieving socket options, specify the level where the option resides and the name of the option.

All applications containing the **getsockopt** subroutine must be compiled with the **_BSD** macro 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
Socket	Specifies the unique socket name.
Level	Specifies the protocol level where the option resides. Options can be retrieved at the following levels: **Socket level** Specifies the *Level* parameter as the **SOL_SOCKET** option. **Other levels** Supplies the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set the *Level* parameter to the protocol number of TCP, as defined in the **netinet/in.h** file.
OptionName	Specifies a single option. The *OptionName* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The **sys/socket.h** file contains definitions for socket level options. The **netinet/tcp.h** file contains definitions for TCP protocol level options. Socket-level options can be enabled or disabled; they operate in a toggle fashion. The **sys/atmsock.h** file contains definitions for ATM protocol level options. The following list defines socket protocol level options found in the **sys/socket.h** file: **SO_DEBUG** Specifies the recording of debugging information. This option enables or disables debugging in the underlying protocol modules. **SO_BROADCAST** Specifies whether transmission of broadcast messages is supported. The option enables or disables broadcast support. **SO_CKSUMREV** Enables performance enhancements in the protocol layers. If the protocol supports this option, enabling causes the protocol to defer checksum verification until the user's data is moved into the user's buffer (on **recv**, **recvfrom**, **read**, or **recvmsg** thread). This can cause applications to be awakened when no data is available, in the case of a checksum error. In this case, EAGAIN is returned. Applications that set this option must handle the EAGAIN error code returned from a receive call. **SO_REUSEADDR** Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port. A particular IP address can only be bound once to the same port. This option enables or disables reuse of local ports. **SO_REUSEADDR** allows an application to explicitly deny subsequent **bind** subroutine to the port/address of the socket with **SO_REUSEADDR** set. This allows an application to block other applications from binding with the **bind** subroutine. **SO_REUSEPORT** Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the **SO_REUSEPORT** socket option. This option enables or disables the reuse of local port/address combinations. **SO_KEEPALIVE** Monitors the activity of a connection by enabling or disabling the periodic transmission of ACK messages on a connected socket. The idle interval time can be designated using the TCP/IP **no** command. Broken connections are discussed in "Understanding Socket Types and Protocols" in *Communications Programming Concepts*.
OptionName (contd)	**SO_DONTROUTE** Indicates outgoing messages should bypass the standard routing facilities. Does not apply routing on outgoing messages. Directs messages to the appropriate network interface according to the network portion of the destination address. This option enables or disables routing of outgoing messages. **SO_LINGER** Lingers on a **close** subroutine if data is present. This option controls the action taken when an unsent messages queue exists for a socket, and a process performs a **close** subroutine on the socket. If the **SO_LINGER** option is set, the system blocks the process during the **close** subroutine until it can transmit the data or until the time expires. If the **SO_LINGER** option is not specified, and a **close** subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible. The **sys/socket.h** file defines the **linger** structure that contains the **l_linger** member for specifying linger time interval. If linger time is set to anything but 0, the system tries to send any messages queued on the socket. The maximum value that the **l_linger** member can be set to is 65535. If the application has requested SPEC1170 compliant behavior by exporting the XPG_SUS_ENV environment variable, the linger time is *n* seconds; otherwise, the linger time is *n*/100 seconds (ticks), where *n* is the value of the **l_linger** member. **SO_OOBINLINE** Leaves received out-of-band data (data marked urgent) in line. This option enables or disables the receipt of out-of-band data. **SO_SNDBUF** Retrieves buffer size information. **SO_RCVBUF** Retrieves buffer size information. **SO_SNDLOWAT** Retrieves send buffer low-water mark information. **SO_RCVLOWAT** Retrieves receive buffer low-water mark information.

Item	Description
OptionName (contd)	**SO_SNDTIMEO** Retrieves time-out information. This option is settable, but currently not used. **SO_RCVTIMEO** Retrieves time-out information. This option is settable, but currently not used. **SO_PEERID** Retrieves the credential information of the process associated with a peer UNIX domain socket. This information includes the process ID, effective user ID, and effective group ID. The `peercred_struct` structure must be used in order to get the credential information. This structure is defined in the `sys/socket.h` file. **SO_ERROR** Retrieves information about error status and clears. **SO_TYPE** Sets the retrieval of a socket type. The following list defines TCP protocol level options found in the **netinet/tcp.h** file: **TCP_CWND_IF** Increases the factor of the TCP congestion window (cwnd) during the congestion avoidance. The value must be in the range 0 - 100 (0 is disable). The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_CWND_DF** Decrease the factor of the TCP cwnd during the congestion avoidance. The value must be in the range 0 - 100 (0 is disable). The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_NOTENTER_SSTART** Avoids reentering the slow start after the retransmit timeout, which might reset the cwnd to the initial window size, instead of the size of the current slow-start threshold (ss_threshold) value or half of the maximum cwnd (max cwnd/2). The values are 1 for enable and 0 for disable. The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_ NOREDUCE_CWND_IN_FRXMT** Not decrease the cwnd size when in the fast retransmit phrase. The values are 1 for enable and 0 for disable. The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_ NOREDUCE_CWND_EXIT_FRXMT** Not decrease the cwnd size when exits the fast retransmit phrase. The values are 1 for enable and 0 for disable. The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_RFC1323** Indicates whether RFC 1323 is enabled or disabled on the specified socket. A non-zero *OptionValue* returned by the **getsockopt** subroutine indicates the RFC is enabled. **TCP_NODELAY** Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default TCP will follow the Nagle algorithm. To disable this behavior, applications can enable **TCP_NODELAY** to force TCP to always send data immediately. A non-zero *OptionValue* returned by the **getsockopt** subroutine indicates **TCP_NODELAY** is enabled. For example, **TCP_NODELAY** should be used when there is an appliciation using TCP for a request/response.
OptionName (contd)	**TCP_NODELAYACK** Specifies if TCP needs to send immediate acknowledgement packets to the sender. If this option is not set, TCP delays sending the acknowledgement packets by up to 200 ms. This allows the acknowledgements to be sent along with the data on a response and minimizes system overhead. Setting this TCP option might cause a slight increase in system overhead, but can result in higher performance for network transfers if the sender is waiting on the receiver's acknowledgements. The following list defines ATM protocol level options found in the **sys/atmsock.h** file: **SO_ATM_PARM** Retrieves all ATM parameters. This socket option can be used instead of using individual sockets options described below. It uses the **connect_ie** structure defined in **sys/call_ie.h** file. **SO_ATM_AAL_PARM** Retrieves ATM AAL (Adaptation Layer) parameters. It uses the **aal_parm** structure defined in **sys/call_ie.h** file. **SO_ATM_TRAFFIC_DES** Retrieves ATM Traffic Descriptor values. It uses the **traffic_desc** structure defined in **sys/call_ie.h** file. **SO_ATM_BEARER** Retrieves ATM Bearer capability information. It uses the **bearer** structure defined in **sys/call_ie.h** file. **SO_ATM_BHLI** Retrieves ATM Broadband High Layer Information. It uses the **bhli** structure defined in **sys/call_ie.h** file. **SO_ATM_BLLI** Retrieves ATM Broadband Low Layer Information. It uses the **blli** structure defined in **sys/call_ie.h** file. **SO_ATM_QoS** Retrieves ATM Ouality Of Service values. It uses the **qos_parm** structure defined in **sys/call_ie.h** file. **SO_ATM_TRANSIT_SEL** Retrieves ATM Transit Selector Carrier. It uses the **transit_sel** structure defined in **sys/call_ie.h** file. **SO_ATM_MAX_PEND** Retrieves the number of outstanding transmit buffers that are permitted before an error indication is returned to applications as a result of a transmit operation. This option is only valid for non best effort types of virtual circuits. **SO_ATM_CAUSE** Retrieves cause for the connection failure. It uses the **cause_t** structure defined in the **sys/call_ie.h** file.

Item	Description
OptionValue	Specifies a pointer to the address of a buffer. The *OptionValue* parameter takes an integer parameter. The *OptionValue* parameter should be set to a nonzero value to enable a Boolean option or to a value of 0 to disable the option. The following options enable and disable in the same manner: • **SO_DEBUG** • **SO_REUSEADDR** • **SO_KEEPALIVE** • **SO_DONTROUTE** • **SO_BROADCAST** • **SO_OOBINLINE** • **TCP_RFC1323**
OptionLength	Specifies the length of the *OptionValue* parameter. The *OptionLength* parameter initially contains the size of the buffer pointed to by the *OptionValue* parameter. On return, the *OptionLength* parameter is modified to indicate the actual size of the value returned. If no option value is supplied or returned, the *OptionValue* parameter can be 0.

Options at other protocol levels vary in format and name.

Item	Description
IP_DONTFRAG	Get current **IP_DONTFRAG** option value.
IP_FINDPMTU	Get current PMTU value.
IP_PMTUAGE	Get current PMTU time out value.

Item	Description
IP_DONTGRAG	Not supported.
IP_FINDPMTU	Get current PMTU value.
IP_PMTUAGE	Not supported.

Item	Description	Value
IPV6_V6ONLY	Determines whether the socket is restricted to IPv6 communications only.	**Option Type:** int (Boolean interpretation)
	Allows the user to determine the outgoing hop limit value for unicast IPv6 packets.	**Option Type:** int
	Allows the user to determine the outgoing hop limit value for multicast IPv6 packets.	**Option Type:** int
	Allows the user to determine the interface being used for outgoing multicast packets.	**Option Type:** unsigned int
	If a multicast datagram is sent to a group that the sending host belongs, a copy of the datagram is looped back by the IP layer for local delivery (if the option is set to 1). If the option is set to 0, a copy is not looped back.	**Option Type:** unsigned int
	Determines whether the destination IPv6 address and arriving interface index of incoming IPv6 packets are being received as ancillary data on UDP and raw sockets.	**Option Type:** int (Boolean interpretation)
	Determines whether the hop limit of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	**Option Type:** int (Boolean interpretation)
	Determines whether the traffic class of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	**Option Type:** int (Boolean interpretation)
	Determines whether the routing header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	**Option Type:** int (Boolean interpretation)
	Determine whether the hop-by-hop options header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	**Option Type:** int (Boolean interpretation)
	Determines whether the destination options header of incoming IPv6 packets is being received as ancillary data on UDP and raw sockets.	**Option Type:** int (Boolean interpretation)

Item	Description	Value
	Determines the source IPv6 address and outgoing interface index for all IPv6 packets being sent on this socket.	**Option Type:** **struct in6_pktinfo** defined in the **netinet/in.h** file.
	Determines the next hop being used for outgoing IPv6 datagrams on this socket.	**Option Type:** **struct sockaddr_in6** defined in the **netinet/in.h** file.
	Determines the traffic class for outgoing IPv6 datagrams on this socket.	**Option Type:** **int**
	Determines the routing header to be used for outgoing IPv6 datagrams on this socket.	**Option Type:** **struct ip6_rthdr** defined in the **netinet/ip6.h** file.
	Determines the hop-by-hop options header to be used for outgoing IPv6 datagrams on this socket.	**Option Type:** **struct ip6_rthdr** defined in the **netinet/ip6.h** file.
	Determines the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will follow a routing header (if present) and will also be used when there is no routing header specified.	**Option Type:** **struct ip6_dest** defined in the **netinet/ip6.h** file.
	Determines the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will precede a routing header (if present). If no routing header is specified, this option will be silently ignored.	**Option Type:** **struct ip6_dest** defined in the **netinet/ip6.h** file.
	Determines how IPv6 path MTU discovery is being controlled for this socket.	**Option Type:** **int**
	Determines whether fragmentation of outgoing IPv6 packets has been disabled on this socket.	**Option Type:** **int** (Boolean interpretation)
	Determines whether IPV6_PATHMTU messages are being received as ancillary data on this socket.	**Option Type:** **int** (Boolean interpretation)
	Gets the address selection preferences for a socket.	**Option Type:** **int** (Boolean interpretation)
	Determines the current Path MTU for a connected socket.	**Option Type:** **struct ip6_mtuinfo** defined in the **netinet/in.h** file.

Item	Description	Value
IPPROTO_ICMPV6	Allows the user to filter ICMPV6 messages by the ICMPV6 type field. If no filter was set, the default kernel filter will be returned.	**Option Type:** The **icmp6_filter** structure defined in the **netinet/icmp6.h** file.

Return Values

Upon successful completion, the **getsockopt** subroutine returns a value of 0.

If the **getsockopt** subroutine is unsuccessful, the subroutine handler performs the following actions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Upon successful completion of the **IPPROTO_IP** option **IP_PMTUAGE** the returns are:

With AIX Version 6.1:

- Positive non-zero OptionValue.

Upon successful completion of TCP protocol sockets option **IP_FINDPMTU** the returns are:

With AIX Version 6.1:

- OptionValue 0 if PMTU discovery (tcp_pmtu_discover) is not enabled/not available.
- Positive non-zero OptionValue if PMTU is available.

Error Codes

Item	Description
EBADF	The *Socket* parameter is not valid.
EFAULT	The address pointed to by the *OptionValue* parameter is not in a valid (writable) part of the process space, or the *OptionLength* parameter is not in a valid part of the process address space.
EINVAL	The *Level*, *OptionName*, or *OptionLength* is invalid.
ENOBUF	Insufficient resources are available in the system to complete the call.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOPROTOOPT	The option is unknown.
EOPNOTSUPP	The option is not supported by the socket family or socket type.
EPERM	The user application does not have the permission to get or to set this socket option. Check the network tunable option.

Examples

The following program fragment illustrates the use of the **getsockopt** subroutine to determine an existing socket type:

```
#include <sys/types.h>
#include <sys/socket.h>
int type;
socklen_t size = sizeof(int);
if(getsockopt(s, SOL_SOCKET, SO_TYPE, (void*)&type,&size)<0){
.
.
.
}
```

Related reference
bind Subroutine
shutdown Subroutine
Related information
no subroutine
Sockets Overview

h

AIX runtime services beginning with the letter *h*.

htonl Subroutine

Purpose

Converts an unsigned long integer from host byte order to Internet network byte order.

Library

ISODE Library (**libisode.a**)

Syntax

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

uint32_t htonl (*HostLong*)
uint32_t *HostLong*;

Description

The **htonl** subroutine converts an unsigned long (32-bit) integer from host byte order to Internet network byte order.

The Internet network requires addresses and ports in network standard byte order. Use the **htonl** subroutine to convert the host integer representation of addresses and ports to Internet network byte order.

The **htonl** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is the same as the network byte order.

The **htonl** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not the same as the network byte order.

All applications containing the **htonl** subroutine must be compiled with the **_BSD** macro 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
HostLong	Specifies a 32-bit integer in host byte order.

Return Values

The **htonl** subroutine returns a 32-bit integer in Internet network byte order (most significant byte first).

Related information
Sockets Overview

htonll Subroutine

Purpose

Converts an unsigned long integer from host byte order to Internet network byte order.

Library

ISODE Library (**libisode.a**)

Syntax

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

uint64_t htonll (*HostLong*)
uint64_t *HostLong*;

Description

The **htonll** subroutine converts an unsigned long (64-bit) integer from host byte order to Internet network byte order.

The Internet network requires addresses and ports in network standard byte order. Use the **htonll** subroutine to convert the host integer representation of addresses and ports to Internet network byte order.

The **htonll** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is the same as the network byte order.

The **htonll** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not the same as the network byte order.

All applications containing the **htonll** subroutine must be compiled with the **_BSD** macro 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
HostLong	Specifies a 64-bit integer in host byte order.

Return Values

The **htonll** subroutine returns a 64-bit integer in Internet network byte order (most significant byte first).

Related information

Sockets Overview

htons Subroutine

Purpose

Converts an unsigned short integer from host byte order to Internet network byte order.

Library

ISODE Library (**libisode.a**)

Syntax

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

uint16_t htons (*HostShort*)
uint16_t *HostShort*;

Description

The **htons** subroutine converts an unsigned short (16-bit) integer from host byte order to Internet network byte order.

The Internet network requires ports and addresses in network standard byte order. Use the **htons** subroutine to convert addresses and ports from their host integer representation to network standard byte order.

The **htons** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is the same as the network byte order.

The **htons** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not the same as the network byte order.

All applications containing the **htons** subroutine must be compiled with the **_BSD** macro 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
HostShort	Specifies a 16-bit integer in host byte order that is a host address or port.

Return Values

The **htons** subroutine returns a 16-bit integer in Internet network byte order (most significant byte first).

Related information
Sockets Overview

i

AIX runtime services beginning with the letter *i*.

if_freenameindex Subroutine

Purpose

Frees the dynamic memory that was allocated by the "if_nameindex Subroutine" on page 195.

Library

Library (**libc.a**)

Syntax

#include <net/if.h>

void if_freenameindex (struct if_nameindex ****ptr*);**

Description

The *ptr* parameter is a pointer returned by the **if_nameindex** subroutine. After the **if_freenameindex** subroutine has been called, the application must not use the array of which *ptr* is the address.

Parameters

Item	Description
ptr	Pointer returned by the **if_nameindex** subroutine

Related information
Subroutines Overview

if_indextoname Subroutine

Purpose

Maps an interface index into its corresponding name.

Library

Standard C Library <**libc.a**>

Syntax

```
#include <net/if.h>
char *if_indextoname(unsigned int ifindex, char *ifname);
```

Description

When the **if_indextoname** subroutine is called, the *ifname* parameter points to a buffer of at least IF_NAMESIZE bytes. The **if_indextoname** subroutine places the name of the interface in this buffer with the *ifindex* index.

Note: IF_NAMESIZE is also defined in <**net/if.h**> and its value includes a terminating null byte at the end of the interface name.

If *ifindex* is an interface index, the **if_indextoname** Subroutine returns the *ifname* value, which points to a buffer containing the interface name. Otherwise, it returns a NULL pointer and sets the **errno** global value to indicate the error.

If there is no interface corresponding to the specified index, the **errno** global value is set to **ENXIO**. If a system error occurs (such as insufficient memory), the **errno** global value is set to the proper value (such as, **ENOMEM**).

Parameters

Item	Description
ifindex	Possible interface index
ifname	Possible name of an interface

Error Codes

Item	Description
ENXIO	There is no interface corresponding to the specified index
ENOMEM	Insufficient memory

Related information
Subroutines Overview

if_nameindex Subroutine

Purpose

Retrieves index and name information for all interfaces.

Library

The Standard C Library (<**libc.a**>)

Syntax

```
#include <net/if.h>

struct if_nameindex *if_nameindex(void)
```

```
struct if_nameindex {
unsigned int if_index; /* 1, 2, ... */
char *if_name; /* null terminated name: "le0", ... */
};
```

Description

The **if_nameindex** subroutine returns an array of **if_nameindex** structures (one per interface).

The memory used for this array of structures is obtained dynamically. The interface names pointed to by the *if_name* members are obtained dynamically as well. This memory is freed by the **if_freenameindex** subroutine.

The function returns a NULL pointer upon error, and sets the **errno** global value to the appropriate value. If successful, the function returns an array of structures. The end of an array of structures is indicated by a structure with an *if_index* value of 0 and an *if_name* value of NULL.

Related information

Subroutines Overview

if_nametoindex Subroutine

Purpose

Maps an interface name to its corresponding index.

Library

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

Syntax

```
#include <net/if.h>
unsigned int if_nametoindex(const char *ifname);
```

Description

If the *ifname* parameter is the name of an interface, the **if_nametoindex** subroutine returns the interface index corresponding to the *ifname* name. If the *ifname* parameter is not the name of an interface, the **if_nametoindex** subroutine returns a 0 and the **errno** global variable is set to the appropriate value.

Parameters

Item	Description
ifname	Possible name of an interface.

Related information

Subroutines Overview

inet_ntop6_zone Subroutine

Purpose

Converts a binary IPv6 address with the possible zone ID into a text string that is suitable for presentation.

Syntax

```
const char
  inet_ntop6_zone (const void src, char dst, size_t size)
```

Description

The **inet_ntop6_zone** subroutine is preferred over the **inet_ntop** subroutine because it can infer the zone ID (defined in Section 11 of RFC 4007) that might be present in the **sin6_scope_id** field of the **sockaddr_in6** structure.

Functionally, this subroutine uses the **inet_ntop** subroutine to generate the textual representation of the address. It appends the ***%zoneid*** suffix to the string if the **sin6_scope_id** field is non-zero.

Parameters

Item	Description
src	Specifies the **sockaddr_in6** structure that contains the address in the **sin6_addr** field and the zone ID in the **sin6_scope_id** field.
dst	Specifies a buffer where the textual representation of the address is stored, and if non-zero, the zone ID is stored.
size	Specifies the size (in bytes) of the buffer pointed to by the **dst** parameter.

Return Values

If successful, a pointer to the buffer containing the converted address is returned. If unsuccessful, NULL is returned. Upon failure, the **errno** global variable is set to ENOSPC if the **size** parameter indicates that the destination buffer is small.

Related information

inet_ntop Subroutines

inet_pton6_zone Subroutine

Purpose

Converts an IPv6 address in its standard text form which might include a zone ID suffix, into its numeric binary form.

Syntax

```
int
inet_pton6_zone (const char *src, void *dst)
```

Description

The **inet_pton6_zone** subroutine is preferred over the **inet_pton** subroutine because it can infer the zone ID suffix (defined in Section 11 of RFC 4007) that might be present in the textual representation of an IPv6 address.

Functionally, this subroutine removes the zone ID, if present, and stores it in the **sin6_scope_id** field of the **sockaddr_in6** structure pointed to by the **dst** parameter. It uses the **inet_pton** subroutine to convert the removed address, and stores it in the **sin6_addr** field.

Parameters

Item	Description
src	The string that contains the textual representation of the address.
dst	A pointer to the **sockaddr_in6** structure where the numeric representation is stored. The zone ID, if present, is stored in the **sin6_scope_id** field, and the address is stored in the **sin6_addr** field.

Return Values

If successful, one is returned. If the input is not a valid IPv6 address, zero is returned.

Related information
inet_pton Subroutine

inet6_is_srcaddr Subroutine

Purpose

Verifies that a given local address meets address selection preferences.

Library

Library **(libc.a)**

Syntax

```
# include <netinet/in.h>
int inet6_is_srcaddr(struct sockaddr_in6 *srcaddr, uint32_t flags);
```

Description

inet6_is_src_addr verifies that a local address corresponds to the set of address selection preference flags specified in flags.

The values of address selection preference flags are:

- IPV6_PREFER_SRC_HOME: prefer addresses reachable from a Home source address
- IPV6_PREFER_SRC_COA: prefer addresses reachable from a Care-of source address
- IPV6_PREFER_SRC_TMP: prefer addresses reachable from a temporary address
- IPV6_PREFER_SRC_PUBLIC: the prefer addresses reachable from a public source address
- IPV6_PREFER_SRC_CGA: the prefer addresses reachable from a Cryptographically Generated Address (CGA) source address
- IPV6_PREFER_SRC_NONCGA: the prefer addresses reachable from a non-CGA source address.

For example:

- To check if srcaddr is a Care-of address, flags must be set to IPV6_PREFER_SRC_COA.
- To check if srcaddr is a CGA and a public address, flags must be set to IPV6_PREFER_SRC_CGA | IPV6_PREFER_SRC_PUBLIC.

Parameters

Item	Description
srcaddr	Points to a sockaddr_in6 structure containing the source address to check

Item	Description
flags	Specifies the address selection preferences.

Return Values

- The subroutine returns 1 when the given address corresponds to a local address and satisfies the address selection preferences.
- The subroutine returns -1 if the given address is not a local address or if flags does not specify one of the valid address selection flag value
- The subroutine returns 0 if the given address is a local address but does not satisfies the address selection preferences

inet6_opt_append Subroutine

Purpose

Returns the updated total length of the extension header.

Syntax

```
int inet6_opt_append(void *extbuf, socklen_t extlen, int offset,
                         uint8_t type, socklen_t len, uint_t align,
                         void **databufp);
```

Description

The **inet6_opt_append** subroutine returns the updated total length of the extension header, taking into account adding an option with length *len* and alignment *align*. If *extbuf* is not NULL, then, in addition to returning the length, the subroutine inserts any needed pad option, initializes the option (setting the type and length fields), and returns a pointer to the location for the option content in *databufp*. After **inet6_opt_append()** has been called, the application can use the *databuf* directly, or use **inet6_opt_set_val()** to specify the content of the option.

Parameters

Item	Description
extbuf	If NULL, **inet6_opt_append** will return only the updated length. If *extbuf* is not NULL, in addition to returning the length, the function inserts any needed pad option, initializes the option (setting the **type** and **length** fields) and returns a pointer to the location for the option content in *databufp*.
extlen	Size of the buffer pointed to by *extbuf*.
offset	The length returned by **inet6_opt_init()** or a previous **inet6_opt_append()**.
type	8-bit option type. Must have a value from 2 to 255, inclusive. (0 and 1 are reserved for the *Pad1* and *PadN* options, respectively.)
len	Length of the option data (excluding the option type and option length fields). Must be a value between 0 and 255, inclusive, and is the length of the option data that follows.

Item	Description
align	Alignment of the option data. Must be a value of 1, 2, 4, or 8. The *align* value can not exceed the value of *len*.
databufp	Specifies the content of the option.

Return Values

Item	Description
-1	Option content does not fit in the extension header buffer.
integer value	Updated total length of the extension header.

inet6_opt_find Subroutine

Purpose

Looks for a specified option in the extension header.

Syntax

```
int inet6_opt_find(void *extbuf, socklen_t extlen, int offset,
                        uint8_t *typep, socklen_t *lenp,
                        void **databufp);
```

Description

The **inet6_opt_find** subroutine is similar to the **inet6_opt_next()** function, except this subroutine lets the caller specify the option type to be searched for, instead of always returning the next option in the extension header.

Parameters

Item	Description
extbuf	Specifies the extension header.
extlen	Size of the buffer pointed to by *extbuf*.
offset	Specifies the position where scanning of the extension buffer can continue. Should either be 0 (for the first option) or the length returned by a previous call to **inet6_opt_next()** or **inet6_opt_find()**.
typep	Stores the option type.
lenp	Stores the length of the option data (excluding the option type and option length fields).
databufp	Points to the data field of the option.

Return Values

The **inet6_opt_find** subroutine returns the updated "previous" total length computed by advancing past the option that was returned and past any options that did not match the type. This returned "previous"

length can then be passed to subsequent calls to **inet6_opt_find()** for finding the next occurrence of the same option type.

Item	Description
–1	The option cannot be located, there are no more options, or the option extension header is malformed.

inet6_opt_finish Subroutine

Purpose

Returns the final length of an extension header.

Syntax

```
int inet6_opt_finish(void *extbuf, socklen_t extlen, int offset);
```

Description

The **inet6_opt_finish** subroutine returns the final length of an extension header, taking into account the final padding of the extension header to make it a multiple of 8 bytes.

Parameters

Item	Description
extbuf	If NULL, **inet6_opt_finish** will only return the final length. If *extbuf* is not NULL, in addition to returning the length, the function initializes the option by inserting a *Pad1* or *PadN* option of the proper length.
extlen	Size of the buffer pointed to by *extbuf*.
offset	The length returned by **inet6_opt_init()** or a previous **inet6_opt_append()**.

Return Values

Item	Description
-1	The necessary pad does not fit in the extension header buffer.
integer value	Final length of the extension header.

inet6_opt_get_val Subroutine

Purpose

Extracts data items of various sizes in the data portion of the option.

Syntax

```
int inet6_opt_get_val(void *databuf, int offset, void *val,
                      socklen_t vallen);
```

Description

The **inet6_opt_get_val** subroutine extracts data items of various sizes in the data portion of the option. It is expected that each field is aligned on its natural boundaries, but the subroutine will not rely on the alignment.

Parameters

Item	Description
databuf	Pointer to the data content returned by **inet6_opt_next()** or **inet6_opt_find()**.
offset	Specifies where in the data portion of the option the value should be extracted. The first byte after the option type and length is accessed by specifying an offset of 0.
val	Pointer to the destination for the extracted data.
vallen	Specifies the size of the data content to be extracted.

Return Values

The **inet6_opt_get_val** subroutine returns the offset for the next field (that is, *offset* + *vallen*), which can be used when extracting option content with multiple fields.

inet6_opt_init Subroutine

Purpose

Returns the number of bytes needed for an empty extension header.

Syntax

```
int inet6_opt_init(void *extbuf, socklen_t extlen);
```

Description

The **inet6_opt_init** subroutine returns the number of bytes needed for the empty extension header (that is, a header without any options).

Parameters

Item	Description
extbuf	Specifies NULL for an empty header. If *extbuf* is not NULL, it initializes the extension header to have the correct length field.
extlen	Specifies the size of the extension header. The value of *extlen* must be a positive value that is a multiple of 8.

Return Values

Item	Description
-1	The value of *extlen* is not a positive (non-zero) multiple of 8.
integer value	Number of bytes needed for an empty extension header.

inet6_opt_next Subroutine

Item	Description
–1	There are no more options or the option extension header is malformed.

Purpose

Parses received option extension headers returning the next option.

Syntax

```
int inet6_opt_next(void *extbuf, socklen_t extlen, int offset,
                        uint8_t *typep, socklen_t *lenp,
                        void **databufp);
```

Description

The **inet6_opt_next** subroutine parses received option extension headers, returning the next option. The next option is returned by updating the *typep*, *lenp*, and *databufp* parameters.

Parameters

Item	Description
extbuf	Specifies the extension header.
extlen	Size of the buffer pointed to by *extbuf*.
offset	Specifies the position where scanning of the extension buffer can continue. Should either be 0 (for the first option) or the length returned by a previous call to **inet6_opt_next()** or **inet6_opt_find()**.
typep	Stores the option type.
lenp	Stores the length of the option data (excluding the option type and option length fields).
databufp	Points to the data field of the option.

Return Values

The **inet6_opt_next** subroutine returns the updated "previous" length computed by advancing past the option that was returned. This returned "previous" length can then be passed to subsequent calls to **inet6_opt_next()**. This function does not return any *PAD1* or *PADN* options.

inet6_opt_set_val Subroutine

Purpose

Inserts data items into the data portion of an option.

Syntax

```
int inet6_opt_set_val(void *databuf, int offset, void *val,
                           socklen_t vallen);
```

Description

The **inet6_opt_set_val** subroutine inserts data items of various sizes into the data portion of the option. The caller must ensure that each field is aligned on its natural boundaries. However, even when the alignment requirement is not satisfied, **inet6_opt_set_val** will just copy the data as required.

Parameters

Item	Description
databuf	Pointer to the data area returned by **inet6_opt_append()**.
offset	Specifies where in the data portion of the option the value should be inserted; the first byte after the option type and length is accessed by specifying an offset of 0.
val	Pointer to the data content to be inserted.
vallen	Specifies the size of the data content to be inserted.

Return Values

The function returns the offset for the next field (that is, *offset* + *vallen*), which can be used when composing option content with multiple fields.

inet6_rth_add Subroutine

Purpose

Adds an IPv6 address to the end of the Routing header being constructed.

Syntax

```
int inet6_rth_add(void *bp, const struct in6_addr *addr);
```

Description

The **inet6_rth_add** subroutine adds the IPv6 address pointed to by *addr* to the end of the Routing header being constructed.

Parameters

Item	Description
bp	Points to the buffer of the Routing header.
addr	Specifies which IPv6 address is to be added.

Return Values

Item	Description
0	Success. The *segleft* member of the Routing Header is updated to account for the new address in the Routing header.
–1	The new address could not be added.

inet6_rth_getaddr Subroutine

Purpose

Returns a pointer to a specific IPv6 address in a Routing header.

Syntax

```
struct in6_addr *inet6_rth_getaddr(const void *bp, int index);
```

Description

The **inet6_rth_getaddr** subroutine returns a pointer to the IPv6 address specified by *index* in the Routing header described by *bp*. An application should first call **inet6_rth_segments()** to obtain the number of segments in the Routing header.

Parameters

Item	Description
bp	Points to the Routing header.
index	Specifies the index of the IPv6 address that must be returned. The value of index must be between 0 and one less than the value returned by **inet6_rth_segments()**.

Return Values

Item	Description
NULL	The **inet6_rth_getaddr** subroutine failed.
Valid pointer	Pointer to the address indexed by *index*.

inet6_rth_init Subroutine

Purpose

Initializes a buffer to contain a Routing header.

Syntax

```
void *inet6_rth_init(void *bp, socklen_t bp_len, int type,
                     int segments);
```

Description

The **inet6_rth_init** subroutine initializes the buffer pointed to by *bp* to contain a Routing header of the specified *type* and sets **ip6r_len** based on the *segments* parameter. *bp_len* is only used to verify that the buffer is large enough. The **ip6r_segleft** field is set to 0; **inet6_rth_add()** increments it.

When the application uses ancillary data, the application must initialize any **cmsghdr** fields. The caller must allocate the buffer, and the size of the buffer can be determined by calling **inet6_rth_space()**.

Parameters

Item	Description
bp	Points to the buffer to be initialized.
bp_len	Size of the buffer pointed to by *bp*.

Item	Description
type	Specifies the type of Routing header to be held.
segments	Specifies the number of addresses within the Routing header.

Return Values

Upon success, the return value is the pointer to the buffer (*bp*), and this is then used as the first argument to the **inet6_rth_add()** function.

Item	Description
NULL	The buffer could not be initialized.

inet6_rth_reverse Subroutine

Purpose

Writes a new Routing header that sends datagrams along the reverse route of a Routing header extension header.

Syntax

```
int inet6_rth_reverse(const void *in, void *out);
```

Description

The **inet6_rth_reverse** subroutine takes a Routing header extension header (pointed to by the first argument) and writes a new Routing header that sends datagrams along the reverse of that route. The function reverses the order of the addresses and sets the *segleft* member in the new Routing header to the number of segments. Both arguments are allowed to point to the same buffer (that is, the reversal can occur in place).

Parameters

Item	Description
in	Points to the original Routing header extension header.
out	Points to the new Routing header route that reverses the route of *in*.

Return Values

Item	Description
0	The reverse Routing header was successfully created.
–1	The reverse Routing header could not be created.

inet6_rth_segments Subroutine

Purpose

Returns the number of segments (addresses) contained in a Routing header.

Syntax

```
int inet6_rth_segments(const void *bp);
```

Description

The **inet6_rth_segments** subroutine returns the number of segments (addresses) contained in the Routing header described by *bp*.

Parameters

Item	Description
bp	Points to the Routing header.

Return Values

Item	Description
0 (or greater)	The number of addresses in the Routing header was returned.
–1	The number of addresses of the Routing header could not be returned.

inet6_rth_space Subroutine

Purpose

Returns the required number of bytes to hold a Routing header.

Syntax

```
socklen_t inet6_rth_space(int type, int segments);
```

Description

The **inet6_rth_space** subroutine returns the number of bytes required to hold a Routing header of the specified *type* containing the specified number of *segments* (addresses). For an IPv6 Type 0 Routing header, the number of segments must be between 0 and 127, inclusive. For an IPv6 Type 2 Routing Header, the number of segments must be 1. The return value is simply the space for the Routing header. When the application uses ancillary data, the application must pass the returned length to **CMSG_SPACE()** in order to determine how much memory is needed for the ancillary data object (including the **cmsghdr** structure).

Note: Although **inet6_rth_space** returns the size of the ancillary data, it does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, so that other ancillary data objects can be added, because all the ancillary data objects must be specified to **sendmsg()** as a single **msg_control** buffer.

Parameters

Item	Description
type	Specifies the type of Routing header to be held.
segments	Specifies the number of addresses within the Routing header.

Return Values

Item	Description
0	Either the type of the Routing header is not supported by this implementation or the number of segments is invalid for this type of Routing header.
length	Determines how much memory is needed for the ancillary data object.

inet_addr Subroutine

Purpose

Converts Internet addresses to Internet numbers.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <arpa/inet.h>

in_addr_t inet_addr ( CharString)
register const char *CharString;
```

Description

The **inet_addr** subroutine converts an ASCII string containing a valid Internet address using dot notation into an Internet address number typed as an unsigned integer value. An example of dot notation is 120.121.5.123. The **inet_addr** subroutine returns an error value if the Internet address notation in the ASCII string supplied by the application is not valid.

Note: Although they both convert Internet addresses in dot notation to Internet numbers, the **inet_addr** subroutine and **inet_network** process ASCII strings differently. When an application gives the **inet_addr** subroutine a string containing an Internet address value without a delimiter, the subroutine returns the logical product of the value represented by the string and 0xFFFFFFFF. For any other Internet address, if the value of the fields exceeds the previously defined limits, the **inet_addr** subroutine returns an error value of -1.

When an application gives the **inet_network** subroutine a string containing an Internet address value without a delimiter, the **inet_network** subroutine returns the logical product of the value represented by the string and 0xFF. For any other Internet address, the subroutine returns an error value of -1 if the value of the fields exceeds the previously defined limits.

All applications containing the **inet_addr** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Sample return values for each subroutine are as follows:

Application String	inet_addr Returns	inet_network Returns
0x1234567890abcdef 0x1234567890abcdef. 256.257.258.259	0x090abcdef 0xFFFFFFFF (= -1) 0xFFFFFFFF (= -1)	0x000000ef 0x0000ef00 0x00010203

The ASCII string for the **inet_addr** subroutine must conform to the following format:

```
string::= field | field delimited_field^1-3 | delimited_field^1-3
delimited_field::= delimiter field | delimiter
delimiter::= .
field::= 0X | 0x | 0Xhexadecimal* | 0x hexadecimal* | decimal* | 0 octal
hexadecimal::= decimal |a|b|c|d|e|f|A|B|C|D|E|F
decimal::= octal |8|9
octal::= 0|1|2|3|4|5|6|7
```

Note:

1. ^n indicates *n* repetitions of a pattern.
2. ^n-m indicates *n* to *m* repetitions of a pattern.
3. * indicates 0 or more repetitions of a pattern, up to environmental limits.
4. The Backus Naur form (BNF) description states the space character, if one is used. *Text* indicates text, not a BNF symbol.

The **inet_addr** subroutine requires an application to terminate the string with a null terminator (0x00) or a space (0x30). The string is considered invalid if the application does not end it with a null terminator or a space. The subroutine ignores characters trailing a space.

The following describes the restrictions on the field values for the **inet_addr** subroutine:

Format	Field Restrictions (in decimal)
a	*Value_a* < 4,294,967,296
a.b	*Value_a* < 256; *Value_b* < 16,777,216
a.b.c	*Value_a* < 256; *Value_b* < 256; *Value_c < 65536*
a.b.c.d	*Value_a* < 256; *Value_b* < 256; *Value_c < 256; Value_d < 256*

Applications that use the **inet_addr** subroutine can enter field values exceeding these restrictions. The subroutine accepts the least significant bits up to an integer in length, then checks whether the truncated value exceeds the maximum field value. For example, if an application enters a field value of 0x1234567890 and the system uses 16 bits per integer, then the **inet_addr** subroutine uses bits 0 -15. The subroutine returns 0x34567890.

Applications can omit field values between delimiters. The **inet_addr** subroutine interprets empty fields as 0.

Note:

1. The **inet_addr** subroutine does not check the pointer to the ASCII string. The user must ensure the validity of the address in the ASCII string.
2. The application must verify that the network and host IDs for the Internet address conform to either a Class A, B, or C Internet address. The **inet_attr** subroutine processes any other number as a Class C address.

Parameters

Item	Description
CharString	Represents a string of characters in the Internet address form.

Return Values

For valid input strings, the **inet_addr** subroutine returns an unsigned integer value comprised of the bit patterns of the input fields concatenated together. The subroutine places the first pattern in the most significant position and appends any subsequent patterns to the next most significant positions.

The **inet_addr** subroutine returns an error value of -1 for invalid strings.

Note: An Internet address with a dot notation value of 255.255.255.255 or its equivalent in a different base format causes the **inet_addr** subroutine to return an unsigned integer value of 4294967295. This value is identical to the unsigned representation of the error value. Otherwise, the **inet_addr** subroutine considers 255.255.255.255 a valid Internet address.

Files

Item	Description
/etc/hosts	Contains host names.
/etc/networks	Contains network names.

Related reference
gethostbyname Subroutine
Related information
Sockets Overview
Understanding Network Address Translation

inet_lnaof Subroutine

Purpose

Returns the host ID of an Internet address.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

int inet_Inaof (*InternetAddr*)
struct in_addr *InternetAddr*;

Description

The **inet_lnaof** subroutine masks off the host ID of an Internet address based on the Internet address class. The calling application must enter the Internet address as an unsigned long value.

All applications containing the **inet_lnaof** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Note: The application must verify that the network and host IDs for the Internet address conform to either a Class A, B, or C Internet address. The **inet_lnaof** subroutine processes any other number as a Class C address.

Parameters

Item	Description
InternetAddr	Specifies the Internet address to separate.

Return Values

The return values of the **inet_lnaof** subroutine depend on the class of Internet address the application provides:

Value	Description
Class A	The logical product of the Internet address and 0x00FFFFFF.
Class B	The logical product of the Internet address and 0x0000FFFF.
Class C	The logical product of the Internet address and 0x000000FF.

Files

Item	Description
/etc/hosts	Contains host names.

Related information
Sockets Overview
Understanding Network Address Translation

inet_makeaddr Subroutine

Purpose

Returns a structure containing an Internet Protocol address based on a network ID and host ID provided by the application.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

struct in_addr inet_makeaddr (*Net*, *LocalNetAddr*)
int *Net*, *LocalNetAddr*;

Description

The **inet_makeaddr** subroutine forms an Internet Protocol (IP) address from the network ID and Host ID provided by the application (as integer types). If the application provides a Class A network ID, the **inet_makeaddr** subroutine forms the IP address using the net ID in the highest-order byte and the logical product of the host ID and 0x00FFFFFF in the 3 lowest-order bytes. If the application provides a Class B network ID, the **inet_makeaddr** subroutine forms the IP address using the net ID in the two highest-order bytes and the logical product of the host ID and 0x0000FFFF in the lowest two ordered bytes. If the application does not provide either a Class A or Class B network ID, the **inet_makeaddr** subroutine forms the IP address using the network ID in the 3 highest-order bytes and the logical product of the host ID and 0x0000FFFF in the lowest-ordered byte.

The **inet_makeaddr** subroutine ensures that the IP address format conforms to network order, with the first byte representing the high-order byte. The **inet_makeaddr** subroutine stores the IP address in the structure as an unsigned long value.

The application must verify that the network ID and host ID for the IP address conform to class A, B, or C. The **inet_makeaddr** subroutine processes any nonconforming number as a Class C address.

The **inet_makeaddr** subroutine expects the **in_addr** structure to contain only the IP address field. If the application defines the **in_addr** structure otherwise, then the value returned in **in_addr** by the **inet_makeaddr** subroutine is undefined.

All applications containing the **inet_makeaddr** subroutine must be compiled with the **_BSD** macro 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
Net	Contains an Internet network number.
LocalNetAddr	Contains a local network address.

Return Values

Upon successful completion, the **inet_makeaddr** subroutine returns a structure containing an IP address.

If the **inet_makeaddr** subroutine is unsuccessful, the subroutine returns a -1.

Files

Item	Description
/etc/hosts	Contains host names.

Related information
Sockets Overview
Understanding Network Address Translation

inet_net_ntop Subroutine

Purpose

Converts between binary and text address formats.

Library

Library (**libc.a**)

Syntax

```
char *inet_net_ntop (af, src, bits, dst, size)
int af;
const void *src;
int bits;
char *dst;
size_t size;
```

Description

This function converts a network address and the number of bits in the network part of the address into the CIDR format ascii text (for example, 9.3.149.0/24). The *af* parameter specifies the family of the address. The *src* parameter points to a buffer holding an IPv4 address if the *af* parameter is AF_INET. The *bits* parameter is the size (in bits) of the buffer pointed to by the *src* parameter. The *dst* parameter points

to a buffer where the function stores the resulting text string. The *size* parameter is the size (in bytes) of the buffer pointed to by the *dst* parameter.

Parameters

Item	Description
af	Specifies the family of the address.
src	Points to a buffer holding and IPv4 address if the *af* parameter is AF_INET.
bits	Specifies the size of the buffer pointed to by the *src* parameter.
dst	Points to a buffer where the resulting text string is stored.
size	Specifies the size of the buffer pointed to by the *dst* parameter.

Return Values

If successful, a pointer to a buffer containing the text string is returned. If unsuccessful, NULL is returned. Upon failure, **errno** is set to EAFNOSUPPORT if the *af* parameter is invalid or ENOSPC if the size of the result buffer is inadequate.

Related information
Subroutines Overview

inet_net_pton Subroutine

Purpose

Converts between text and binary address formats.

Library

Library (**libc.a**)

Syntax

```
int inet_net_pton (af, src, dst, size)
int af;
const char *src;
void *dst;
size_t size;
```

Description

This function converts a network address in ascii into the binary network address. The ascii representation can be CIDR-based (for example, 9.3.149.0/24) or class-based (for example, 9.3.149.0). The *af* parameter specifies the family of the address. The *src* parameter points to the string being passed in. The *dst* parameter points to a buffer where the function will store the resulting numeric address. The *size* parameter is the size (in bytes) of the buffer pointed to by the *dst* parameter.

Parameters

Item	Description
af	Specifies the family of the address.
src	Points to the string being passed in.
dst	Points to a buffer where the resulting numeric address is stored.

Item	Description
size	Specifies the size (in bytes) of the buffer pointed to by the *dst* parameter.

Return Values

If successful, the number of bits, either inputted classfully or specified with */CIDR*, is returned. If unsuccessful, a -1 (negative one) is returned (check errno). ENOENT means it was not a valid network specification.

Related information
Subroutines Overview

inet_netof Subroutine

Purpose

Returns the network id of the given Internet address.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_netof ( InternetAddr)
struct in_addr InternetAddr;
```

Description

The **inet_netof** subroutine returns the network number from the specified Internet address number typed as unsigned long value. The **inet_netof** subroutine masks off the network number and the host number from the Internet address based on the Internet address class.

All applications containing the **inet_netof** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Note: The application assumes responsibility for verifying that the network number and the host number for the Internet address conforms to a class A or B or C Internet address. The **inet_netof** subroutine processes any other number as a class C address.

Parameters

Item	Description
InternetAddr	Specifies the Internet address to separate.

Return Values

Upon successful completion, the **inet_netof** subroutine returns a network number from the specified long value representing the Internet address. If the application gives a class A Internet address, the **inet_lnoaf** subroutine returns the logical product of the Internet address and `0xFF000000`. If the application gives a class B Internet address, the **inet_lnoaf** subroutine returns the logical product of the Internet address

and 0xFFFF0000. If the application does not give a class A or B Internet address, the **inet_lnoaf** subroutine returns the logical product of the Internet address and 0xFFFFFF00.

Files

Item	Description
/etc/hosts	Contains host names.
/etc/networks	Contains network names.

Related information
Sockets Overview
Understanding Network Address Translation

inet_network Subroutine

Purpose

Converts an ASCII string containing an Internet network addressee in . (dot) notation to an Internet address number.

Library

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

Syntax

```
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

in_addr_t inet_network (*CharString***)**
register const char ****CharString*;**

Description

The **inet_network** subroutine converts an ASCII string containing a valid Internet address using . (dot) notation (such as, 120.121.122.123) to an Internet address number formatted as an unsigned integer value. The **inet_network** subroutine returns an error value if the application does not provide an ASCII string containing a valid Internet address using . notation.

The input ASCII string must represent a valid Internet address number, as described in "TCP/IP addressing" in *Networks and communication management*. The input string must be terminated with a null terminator (0x00) or a space (0x30). The **inet_network** subroutine ignores characters that follow the terminating character.

The input string can express an Internet address number in decimal, hexadecimal, or octal format. In hexadecimal format, the string must begin with 0x. The string must begin with 0 to indicate octal format. In decimal format, the string requires no prefix.

Each octet of the input string must be delimited from another by a period. The application can omit values between delimiters. The **inet_network** subroutine interprets missing values as 0.

The following examples show valid strings and their output values in both decimal and hexadecimal notation:

Examples of valid strings		
Input String	**Output Value (in decimal)**	**Output Value (in hex)**
...1	1	0x00000001
.1..	65536	0x00010000
1	1	0x1
0xFFFFFFFF	255	0x000000FF
1.	256	0x100
1.2.3.4	66048	0x010200
0x01.0X2.03.004	16909060	0x01020304
1.2. 3.4	16777218	0x01000002
9999.1.1.1	251724033	0x0F010101

The following examples show invalid input strings and the reasons they are not valid:

Examples of invalid strings	
Input String	**Reason**
1.2.3.4.5	Excessive fields.
1.2.3.4.	Excessive delimiters (and therefore fields).
1,2	Bad delimiter.
1p	String not terminated by null terminator nor space.
{empty string}	No field or delimiter present.

Typically, the value of each octet of an Internet address cannot exceed 246. The **inet_network** subroutine can accept larger values, but it uses only the eight least significant bits for each field value. For example, if an application passes `0x1234567890.0xabcdef`, the **inet_network** subroutine returns `37103` (`0x000090EF`).

The application must verify that the network ID and host ID for the Internet address conform to class A, class B, or class C. The **inet_makeaddr** subroutine processes any nonconforming number as a class C address.

The **inet_network** subroutine does not check the pointer to the ASCII input string. The application must verify the validity of the address of the string.

All applications containing the **inet_network** subroutine must be compiled with the **_BSD** macro 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	Represents a string of characters in the Internet address form.

Return Values

For valid input strings, the **inet_network** subroutine returns an unsigned integer value that comprises the bit patterns of the input fields concatenated together. The **inet_network** subroutine places the first pattern in the leftmost (most significant) position and appends subsequent patterns if they exist.

For invalid input strings, the **inet_network** subroutine returns a value of -1.

Files

Item	Description
/etc/hosts	Contains host names.
/etc/networks	Contains network names.

Related information
Sockets Overview
Understanding Network Address Translation

inet_ntoa Subroutine

Purpose

Converts an Internet address into an ASCII string.

Library

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

Syntax

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

char *inet_ntoa (*InternetAddr***)**
struct in_addr *InternetAddr***;**

Description

The **inet_ntoa** subroutine takes an Internet address and returns an ASCII string representing the Internet address in dot notation. All Internet addresses are returned in network order, with the first byte being the high-order byte.

Use C language integers when specifying each part of a dot notation.

All applications containing the **inet_ntoa** subroutine must be compiled with the **_BSD** macro 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
InternetAddr	Contains the Internet address to be converted to ASCII.

Return Values

Upon successful completion, the **inet_ntoa** subroutine returns an Internet address.

If the **inet_ntoa** subroutine is unsuccessful, the subroutine returns a -1.

Files

Item	Description
/etc/hosts	Contains host names.
/etc/networks	Contains network names.

Related information
Sockets Overview
Understanding Network Address Translation

inet_ntop Subroutine

Purpose

This function is deprecated for AF_INET6 in favor of the inet_ntop6_zone Subroutine .

Library

Library (**libc.a**)

Syntax

```
const char *inet_ntop (af, src, dst, size)
int af;
const void *src;
char *dst;
size_t size;
```

Description

This function converts from an address in binary format (as specified by the *src* parameter) to standard text format, and places the result in the *dst* parameter (if *size*, which specifies the space available in the *dst* parameter, is sufficient). The *af* parameter specifies the family of the address. This can be AF_INET or AF_INET6.

The *src* parameter points to a buffer holding an IPv4 address if the *af* parameter is AF_INET, or an IPv6 address if the *af* parameter is AF_INET6. The *dst* parameter points to a buffer where the function will store the resulting text string. The *size* parameter specifies the size of this buffer (in bytes). The application must specify a non-NULL *dst* parameter. For IPv6 addresses, the buffer must be at least INET6_ADDRSTRLEN bytes. For IPv4 addresses, the buffer must be at least INET_ADDRSTRLEN bytes.

In order to allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in the <**netinet/in.h**> library:

```
#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

Parameters

Item	Description
af	Specifies the family of the address. This can be AF_INET or AF_INET6.
src	Points to a buffer holding an IPv4 address if the *af* parameter is set to AF_INET, or an IPv6 address if the *af* parameter is set to AF_INET6.
dst	Points to a buffer where the resulting text string is stored.
size	Specifies the size (in bytes) of the buffer pointed to by the *dst* parameter.

Return Values

If successful, a pointer to the buffer containing the converted address is returned. If unsuccessful, NULL is returned. Upon failure, the **errno** global variable is set to EAFNOSUPPORT if the specified address family (*af*) is unsupported, or to ENOSPC if the *size* parameter indicates the destination buffer is too small.

Related information
Subroutines Overview

inet_pton Subroutine

Purpose

This function is deprecated for AF_INET6 in favor of the**inet_pton6_zone Subroutine** .

Library

Library (**libc.a**)

Syntax

```
int inet_pton (af, src, dst)
int af;
const char *src;
void *dst;
```

Description

This function converts an address in its standard text format into its numeric binary form. The *af* parameter specifies the family of the address.

Note: Only the AF_INET and AF_INET6 address families are supported.

Parameters

Item	Description
af	Specifies the family of the address. This can be AF_INET or AF_INET6.
src	Points to the string being passed in.
dst	Points to a buffer where the function stores the numeric address. The address is returned in network byte order.

Return Values

If successful, one is returned. If unsuccessful, zero is returned if the input is not a valid IPv4 dotted-decimal string or a valid IPv6 address string; or a negative one with the **errno** global variable set to EAFNOSUPPORT if the *af* parameter is unknown. The calling application must ensure that the buffer referred to by the *dst* parameter is large enough to hold the numeric address (4 bytes for AF_INET or 16 bytes for AF_INET6).

If the *af* parameter is AF_INET, the function accepts a string in the standard IPv4 dotted-decimal form.

```
ddd.ddd.ddd.ddd
```

Where *ddd* is a one to three digit decimal number between 0 and 255.

Note: Many implementations of the existing **inet_addr** and **inet_aton** functions accept nonstandard input such as octal numbers, hexadecimal numbers, and fewer than four numbers. **inet_pton** does not accept these formats.

If the *af* parameter is AF_INET6, then the function accepts a string in one of the standard IPv6 text forms defined in the addressing architecture specification.

Related information
Subroutines Overview

innetgr, getnetgrent, setnetgrent, or endnetgrent Subroutine

Purpose

Handles the group network entries.

Library

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

Syntax

```
#include <netdb.h>
```

innetgr (*NetGroup*, *Machine*, *User*, *Domain*)
char * *NetGroup*, * *Machine*, * *User*, * *Domain*;

getnetgrent (*MachinePointer*, *UserPointer*, *DomainPointer*)
char ** *MachinePointer*, ** *UserPointer*, ** *DomainPointer*;

```
void setnetgrent (NetGroup)
char *NetGroup
```

```
void endnetgrent ()
```

Description

The **innetgr** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **innetgr** subroutine returns *1* or *0*, depending on if **netgroup** contains the *machine*, *user*, *domain* triple as a member. Any of these three strings; *machine*, *user*, or *domain*, can be NULL, in which case it signifies a wild card.

The **getnetgrent** subroutine returns the next member of a network group. After the call, *machinepointer* will contain a pointer to a string containing the name of the machine part of the network group member, and similarly for *userpointer* and *domainpointer*. If any of *machinepointer*, *userpointer*, or *domainpointer* is returned as a NULL pointer, it signifies a wild card. The **getnetgrent** subroutine uses malloc to allocate space for the name. This space is released when the **endnetgrent** subroutine is called. **getnetgrent** returns *1* if it succeeded in obtaining another member of the network group or 0 when it has reached the end of the group.

The **setnetgrent** subroutine establishes the network group from which the **getnetgrent** subroutine will obtain members, and also restarts calls to the **getnetgrent** subroutine from the beginnning of the list. If the previous **setnetgrent()** call was to a different network group, an **endnetgrent()** call is implied. **endnetgrent()** frees the space allocated during the **getnetgrent()** calls.

Parameters

Item	Description
Domain	Specifies the domain.
DomainPointer	Points to the string containing *Domain* part of the network group.

Item	Description
Machine	Specifies the machine.
MachinePointer	Points to the string containing *Machine* part of the network group.
NetGroup	Points to a network group.
User	Specifies a user.
UserPointer	Points to the string containing *User* part of the network group.

Return Values

Item	Description
1	Indicates that the subroutine was successful in obtaining a member.
0	Indicates that the subroutine was not successful in obtaining a member.

Files

Item	Description
/etc/netgroup	Contains network groups recognized by the system.
/usr/include/netdb.h	Contains the network database structures.

Related information
Sockets Overview

ioctl Socket Control Operations

Purpose

Performs network-related control operations.

Syntax

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

int ioctl (fd,  cmd, .../* arg */)
int  fd;
int  cmd;
int ...  /* arg */
```

Description

The socket ioctl commands does various network-related control. The *fd* argument is a socket descriptor. For non-socket descriptors, the functions that are performed by this call are unspecified.

The *cmd* argument and an optional third argument (with varying type) are passed to and interpreted by the socket ioctl function to perform an appropriate control operation that is specified by the user.

The socket ioctl control operations can be in the following control operations categories:

- Socket
- Routing table
- ARP table
- Global network parameters

- Interface

Parameters

Item	Description
fd	Open file descriptor that refers to a socket created by using socket or accept calls.
cmd	Selects the control function to be performed.
.../* *arg* */	Represents information that is required for the requested function. The type of *arg* depends on the particular control request, but it is either an integer or a pointer to a socket-specific data structure.

Socket Control Operations

The following ioctl commands operate on sockets:

ioctl command	Description

SIOCATMARK	Determines whether the read pointer is pointing to the logical mark in the DataStream. The logical mark indicates the point at which the out-of-band data is sent.
	`ioctl(fd, SIOCATMARK,&atmark);`
	`int atmark;`
	If *atmark* is set to 1 on return, the read pointer points to the mark and the next read returns data after the mark. If *atmark* is set to 0 on return (assuming out-of-band data is present on the DataStream), the next read returns data that is sent before the out-of-band mark.
	Note: The out-of-band data is a logically independent data channel that is delivered to the user independently of normal data; in addition, a signal is also sent because of the immediate attention required. Ctrl-C characters are an example.

SIOCSPGRP
SIOCGPGRP
	SIOCSPGRP sets the process group information for a socket. SIOCGPGRP gets the process group ID associated with a socket.
	`ioctl (fd, cmd, (int)&pgrp);`
	`int  pgrp;`
	cmd
	Set to SIOCSPGRP or SIOCGPGRP.
	pgrp
	Specifies the process group ID for the socket.

Routing Table Control Operations

The following ioctl commands operate on the kernel routing table:

ioctl command	Description

SIOCADDRT
SIOCDELRT
	SIOCADDRT adds a route entry in the routing table. SIOCDELRT deletes a route entry from the routing table.
	`ioctl(fd, cmd, (caddr_t)&route);`
	`struct ortentry route;`
	cmd
	Set to SIOCADDRT or SIOCDELRT.
	The route entry information is passed in the `ortentry` structure.

SIOUPDROUTE	Updates the routing table by using the information that is passed in the `ifreq` structure.
	`ioctl (fd, SIOUPDROUTE, (caddr_t)&ifr);`
	`struct ifreq  ifr;`

ARP Table Control Operations

The following ioctl commands operate on the kernel ARP table. The `net/if_arp.h` header file must be included.

ioctl command	Description

SIOCSARP
SIOCDARP
SIOCGARP
	SIOCSARP adds or modifies an ARP entry in the ARP table. SIOCDARP deletes an ARP entry from the ARP table. SIOCGARP gets an ARP entry from the ARP table.
	`ioctl(fd, cmd, (caddr_t)&ar);`
	`struct arpreq ar;`
	cmd
	Set to SIOCSARP, SIOCDARP, or SIOCGARP.
	The ARP entry information is passed in the `arpreq` structure. If `ar.if Type = IFT_IB` and the command is SIOCDARP, the InfiniBand (IB) ARP entry is deleted.

Global Network Parameters Control Operations

The following ioctl commands operate as global network parameters:

ioctl command	Description

SIOCSNETOPT
SIOCGNETOPT
SIOCDNETOPT
SIOCGNETOPT1
	SIOCSNETOPT sets the value of a network option. SIOCGNETOPT gets the value of a network option. SIOCDNETOPT sets the default values of a network option.
	`ioctl(fd, cmd, (caddr_t)&oreq);`
	`struct optreq oreq;`
	cmd
	Set to SIOCSNETOPT, SIOCGNETOPT, or SIOCDNETOPT.
	The network option value is stored in the `optreq` structure.
	SIOCGNETOPT1 gets the current value, default value, and the range of a network option.
	`ioctl(fd, SIOCGNETOPT1, (caddr_t)&oreq);`
	`struct optreq1 oreq;`
	The network option information is stored in the `optreq1` structure upon return The `optreq` and `optreq1` structures are defined in `net/netopt.h`.

SIOCGNMTUS
SIOCGETMTUS
SIOCADDMTU
SIOCDELMTU
	SIOCGNMTUS gets the number of MTUs maintained in the list of common MTUs. SIOCADDMTU adds an MTU in the list of common MTUs. SIOCDELMTU deletes an MTU from the list of common MTUs.
	`ioctl(fd, cmd, (caddr_t)&nmtus);`
	`int nmtus;`
	cmd
	Set to SIOCGNMTUS, SIOCADDMTU, or SIOCDELMTU.
	SIOCGETMTUS gets the MTUs maintained in the list of common MTUs.
	`ioctl(fd, SIOCGETMTUS, (caddr_t)&gm);`
	`struct get_mtus gm;`
	The `get_mtus` structure is defined in `netinet/in.h`.

Interface Control Operations

The following ioctl commands operate on interfaces. The `net/if.h` header file must be included.

ioctl command	Description

SIOCSIFADDR
SIOCDIFADDR
	SIOCSIFADDR sets an interface address. SIOCDIFADDR deletes an interface address. The interface address is specified in the *ifr.ifr_addr* field. SIOCGIFADDR gets an interface address. The address is returned in the *ifr.ifr_addr* field.
	`ioctl(fd, cmd, (caddr_t)&ifr, sizeof(struct ifreq));`
	`struct ifreq ifr;`
	cmd
	Set to SIOCSIFADDR, or SIOCDIFADDR.

ioctl command	Description

SIOCAIFADDR	SIOCAIFADDR adds an interface address. The interface name is specified in the *ifr.ifra_name* field. The alias IP address is specified in the *theifr.ifra_addr* field. The alias IP broadcast address might be specified in the *ifr.ifra_broadaddr* field, and the alias IP network mask might be specified in the *ifr.ifra_mask*.
	`ioctl(fd, cmd, (caddr_t)&ifr, sizeof(struct ifaliasreq));`
	`struct ifaliasreq ifr;`
	cmd
	Set to SIOCAIFADDR

SIOCGIFADDRS	Gets the list of addresses that are associated with an interface.
	`ioctl (fd, SIOCGIFADDRS, (caddr_t)ifaddrsp);`
	`struct ifreqaddrs  *ifaddrsp;`
	The interface name is passed in the *ifaddrsp->ifr_name field*. The addresses that are associated with the interface are stored in `ifaddrsp->ifrasu` array on return.
	Note: The `ifreqaddrs` structure contains space for storing only one `sockaddr_in/sockaddr_in6` structure (array of one `sockaddr_in/sockaddr_in6` element). To get *n* addresses associated with an interface, the caller of the ioctl command must allocate space for *{sizeof (struct ifreqaddrs) + (n * sizeof (struct sockaddr_in)}* bytes.

SIOCSIFDSTADDR
SIOCGIFDSTADDR
	SIOCSIFDSTADDR sets the point-to-point address for an interface that is specified in the *ifr.ifr_dstaddr* field. SIOCGIFDSTADDR gets the point-to-point address that is associated with an interface. The address is stored in the *ifr.ifr_dstaddr* field on return.
	`ioctl(fd, cmd, (caddr_t)&ifr, sizeof(struct ifreq));`
	`struct ifreq ifr;`
	cmd
	Set to SIOCSIFDSTADDR or SIOCGIFDSTADDR.

SIOCSIFNETMASK
SIOCGIFNETMASK
	SIOCSIFNETMASK sets the interface netmask that is specified in the *ifr.ifr_addr* field. SIOCGIFNETMASK gets the interface netmask.
	`ioctl(fd, cmd, (caddr_t)&ifr, sizeof(struct ifreq));`
	`struct ifreq ifr;`
	cmd
	Set to SIOCSIFNETMASK or SIOCGIFNETMASK.

SIOCSIFBRDADDR
SIOCGIFBRDADDR
	SIOCSIFBRDADDR sets the interface broadcast address that is specified in the *ifr.ifr_broadaddr* field. SIOCGIFBRDADDR gets the interface broadcast address. The broadcast address is placed in the *ifr.ifr_broadaddr* field.
	`ioctl(fd, cmd, (caddr_t)&ifr, sizeof(struct ifreq));`
	`struct ifreq ifr;`
	cmd
	Set to SIOCSIFBRDADDR or SIOCGIFBRDADDR.

ioctl command	Description

SIOCGSIZIFCONF	Gets the size of memory that is required to get configuration information for all interfaces returned by SIOCGIFCONF.
	`ioctl(fd, cmd, (caddr_t)&ifconfsize);`
	`int ifconfsize;`

SIOCGIFCONF	Returns configuration information for all the interfaces that are configured on the system.
	`ioctl(fd, SIOCGIFCONF, (caddr_t)&ifc);`
	`struct ifconf ifc;`
	The configuration information is returned in a list of `ifreq` structures pointed to by the *ifc.ifc_req* field, with one `ifreq` structure per interface.
	Note: The caller of the ioctl command must allocate sufficient space to store the configuration information, returned as a list of `ifreq` structures for all of the interfaces that are configured on the system. For example, if *n* interfaces are configured on the system, *ifc.ifc_req* must point to *{n * sizeof (struct ifreq)}* bytes of space allocated.
	Note: Alternatively, the SIOCGSIZIFCONF ioctl command can be used for this purpose.

SIOCSIFFLAGS
SIOCGIFFLAGS
	SIOCSIFFLAGS sets the interface flags. SIOCGIFFLAGS gets the interface flags.
	`ioctl(fd, cmd, (caddr_t)&ifr);`
	`struct ifreq ifr;`
	Refer to `/usr/include/net/if.h` for the interface flags, denoted by `IFF_`*xxx*.
	Note: The `IFF_BROADCAST`, `IFF_POINTTOPOINT`, `IFF_SIMPLEX`, `IFF_RUNNING`, `IFF_OACTIVE`, and `IFF_MULTICAST` flags cannot be changed by using ioctl.

SIOCSIFMETRIC
SIOCGIFMETRIC
	SIOCSIFMETRIC sets the interface metric that is specified in the *ifr.ifr_metric* field. SIOCGIFMETRIC gets the interface metric. The interface metric is placed in the *ifr.ifr_metric* field on return.
	`ioctl(fd, cmd, (caddr_t)&ifr);`
	`struct ifreq ifr;`
	cmd
	Set to SIOCSIFMETRIC or SIOCGIFMETRIC.

SIOCSIFSUBCHAN
SIOCGIFSUBCHAN
	SIOCSIFSUBCHAN sets the subchannel address that is specified in the *ifr.ifr_flags* field. SIOCGIFSUBCHAN gets the subchannel address in the *ifr.ifr_flags* field.
	`ioctl(fd, SIOCSIFSUBCHAN, (caddr_t)&ifr);`
	`struct ifreq ifr;`

ioctl command	Description

SIOCSIFOPTIONS
SIOCGIFOPTIONS
	SIOCSIFOPTIONS sets the interface options. SIOCGIFOPTIONS gets the interface options.
	`ioctl(fd, SIOCSIFOPTIONS, (caddr_t)&ifr);`
	`struct ifreq ifr;`
	The interface options are stored in the *ifr_flags* field of the `ifreq` structure. Refer to `/usr/include/net/if.h` file for the list of interface options that are denoted by `IFO_xxx`.

ioctl command	Description

SIOCADDMULTI
SIOCDELMULTI
	SIOCADDMULTI adds an address to the list of multicast addresses for an interface. SIOCDELMULTI deletes a multicast address from the list of multicast addresses for an interface.
	`ioctl(fd, cmd, (caddr_t)&ifr);`
	`struct ifreq ifr;`
	cmd
	Set to SIOCADDMULTI or SIOCDELMULTI.
	The multicast address information is specified in the `ifr_addr` structure.

SIOCGETVIFCNT	Gets the packet count information for a virtual interface. The information is specified in the `sioc_vif_req` structure.
	`ioctl (fd, SIOCGETVIFCNT, (caddr_t)&v_req);`
	`struct sioc_vif_req  v_req;`

SIOCGETSGCNT	Gets the packet count information for the source group specified. The information is stored in the `sioc_sg_req` structure on return.
	`ioctl(fd, SIOCGETSGCNT, (caddr_t)&v_req);`
	`struct sioc_sg_req v_req;`

SIOCSIFMTU
SIOCGIFMTU
	SIOCSIFMTU sets the interface maximum transmission unit (MTU). SIOCGIFMTU gets the interface MTU.
	`ioctl(fd, cmd, (caddr_t)&ifr);`
	`struct ifreq ifr;`
	The MTU value is stored in *ifr.ifr_mtu* field.
	Note: The range of valid values for MTU varies for an interface and is dependent on the interface type.

SIOCIFATTACH
SIOCIFDETACH
	SIOCIFATTACH attaches an interface. This initializes and adds an interface in the network interface list. SIOCIFDETACH detaches an interface broadcast address. This removes the interface from the network interface list. The interface name is specified in the *ifr.ifr_name* field.
	`ioctl(fd, cmd, (caddr_t)&ifr);`
	`struct ifreq ifr;`

ioctl command	Description

SIOCSIFGIDLIST
SIOCGIFGIDLIST
	SIOCSIFGIDLIST adds or deletes the list of group IDs specified in the *ifrg.ifrg_gidlist* field to the *gidlist* interface. The interface name is specified in the *ifrg.ifrg_name* field. An operation code, ADD_GRP/DEL_GRP, specified in the *ifrg.ifrg_gidlist* field indicates whether the specified list of group IDs must be added to or deleted from the *gidlist* interface. SIOCGIFGIDLIST gets the list of group IDs associated with an interface. The group IDs are placed in the *ifrg.ifrg_gidlist* field on return.
	`ioctl(fd, cmd, (caddr_t)&ifrg);`
	`struct ifgidreq ifrg;`

SIOCIF_ATM_UBR SIOCIF_ATM_SNMPARP SIOCIF_ATM_DUMPARP SIOCIF_ATM_IDLE SIOCIF_ATM_SVC SIOCIF_ATM_DARP SIOCIF_ATM_GARP SIOCIF_ATM_SARP	SIOCIF_ATM_UBR sets the UBR rate for an ATM interface. SIOCIF_ATM_SNMPARP gets the SNMP ATM ARP entries. SIOCIF_ATM_DUMPARP gets the specified number of ATM ARP entries. SIOCIF_ATM_DARP deletes an ATM ARP entry from the ARP table. SIOCIF_ATM_GARP gets an ATM ARP entry to the ARP table. SIOCIF_ATM_SARP adds an ATM ARP entry. The ARP information is specified in the `atm_arpreq` structure. SIOCIF_ATM_SVC specifies whether this interface supports Permanent Virtual Circuit (PVC) and Switched Virtual Circuit (SVC) types of virtual connections. It also specifies whether this interface is an ARP client or an ARP server for this Logical IP Subnetwork (LIS) based on the flag that is set in the `ifatm_svc_arg` structure. SIOCIF_ATM_IDLE specifies the idle time limit on the interface.

SIOCSISNO
SIOCGISNO
	SIOCSISNO sets interface specific network options for an interface. SIOCGISNO gets interface specific network options that are associated with an interface.
	`ioctl(fd, cmd, (caddr_t)&ifr);`
	`struct ifreq ifr;`
	cmd
	Set to SIOCSISNO or SIOCGISNO.
	The interface-specific network options are stored in `ifr.ifr_isno` structure. Refer to `/usr/include/net/if.h` file for the list of interface-specific network options that are denoted by `ISNO_`*xxx*.

SIOCGIFBAUDRATE	Gets the value of the interface baud rate in the *ifr_baudrate* field.
	`ioctl(fd, SIOCGIFBAUDRATE, (caddr_t)&ifr);`
	`struct ifreq ifr;`
	The baud rate is stored in the *ifr.ifr_baudrate* field.

ioctl command	Description

SIOCADDIFVIPA
SIOCDELIFVIPA
SIOCLISTIFVIPA
	SIOCADDIFVIPA associates the specified list of interfaces pointed to by *ifrv.ifrv_ifname* with the virtual interface specified by *ifrv.ifrv_name*. This operation causes the source address for all outgoing packets on these interfaces to be set to the virtual interface address. SIOCDELIFVIPA removes the list of specified interfaces that are pointed by *ifrv.ifrv_ifname* and associated with the virtual interface specified by *ifrv.ifrv_name*, by using SIOCADDIFVIPA. SIOCLISTIFVIPA lists all the interfaces that are associated with the virtual interface specified by *ifrv.ifrv_name*.
	`ioctl(fd, SIOCADDIFVIPA, (caddr_t)&ifrv);`
	`struct ifvireq ifrv;`
	The virtual interface information is stored in the `ifvireq` structure.
	Note: These flags operate on a virtual interface only.

SIOCSIFADDR6	Set or Add an IPv6 address.
	`ioctl(fd, SIOCSIFADDR6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

SIOCGIFADDR6	Gets an IPv6 address.
	`ioctl(fd, SIOCGIFADDR6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

SIOCSIFDSTADDR6	Set the destination (point-to-point) address for a IPv6 address.
	`ioctl(fd, SIOCSIFDSTADDR6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

SIOCGIFDSTADDR6	Get the destination (point-to-point) address for a IPv6 address.
	`ioctl(fd, SIOCGIFDSTADDR6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

SIOCSIFNETMASK6	Set the netmask for an IPv6 address.
	`ioctl(fd, SIOCSIFNETMASK6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

SIOCGIFNETMASK6	Get the netmask for an IPv6 address.
	`ioctl(fd, SIOCGIFNETMASK6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

SIOCDIFADDR6	Delete an IPv6 address.
	`ioctl(fd, SIOCDIFADDR6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

SIOCFIFADDR6	Put an IPv6 address at the beginning of the address list.
	`ioctl(fd, SIOCFIFADDR6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

ioctl command	Description

SIOCAIFADDR6	Add or change an IPv6 alias address.
	`ioctl(fd, SIOCAIFADDR6, (caddr_t)&ifra);`
	`struct in6_aliasreq ifra;`

SIOCADDANY6	Add an IPv6 anycast address.
	`ioctl(fd, SIOCADDANY6, (caddr_t)&ifra);`
	`struct in6_ifreq ifr;`

SIOCDELANY6	Delete an IPv6 anycast address.
	`ioctl(fd, SIOCDELANY6, (caddr_t)&ifra);`
	`struct in6_ifreq ifr;`

SIOCSIFZONE6	Set the IPv6 zone ID of an interface at a particular address scope.
	`ioctl(fd, SIOCSIFZONE6, (caddr_t)&ifrz);`
	`struct in6_zonereq ifrz;`

SIOCGIFZONE6	Get the IPv6 scope zone IDs of an interface.
	`ioctl(fd, SIOCGIFZONE6, (caddr_t)&ifrz);`
	`struct in6_zonereq ifrz;`

SIOCSIFADDRORI6	Set the configuration origin for an IPv6 address.
	`ioctl(fd, SIOCSIFADDRORI6, (caddr_t)&ifro);`
	`struct ifaddrorigin6 ifro;`

SIOCAIFADDR6T	Add or change an IPv6 alias address and type.
	`ioctl(fd, SIOCAIFADDR6T, (caddr_t)&ifra);`
	`struct in6_aliasreq2 ifra;`

SIOCGIFADDR6T	Get the type of an IPv6 address.
	`ioctl(fd, SIOCGIFADDR6T, (caddr_t)&ifra);`
	`struct in6_aliasreq2 ifra;`

SIOCSIFADDRSTATE6	Change the state of an IPv6 address.
	`ioctl(fd, SIOCSIFADDRSTATE6, (caddr_t)&ifra);`
	`struct in6_aliasreq2 ifra;`

SIOCGIFADDRSTATE6	Get the state of an IPv6 address.
	`ioctl(fd, SIOCGIFADDRSTATE6, (caddr_t)&ifra);`
	`struct in6_aliasreq2 ifra;`

SIOCGSRCFILTER6	Get the IPv6 multicast group source filter for an interface.
	`ioctl(fd, SIOCGSRCFILTER6, (caddr_t)&ifrgsf);`
	`struct group_source_filter_req ifrgsf;`

SIOCACLADDR6	Add an IPv6 cluster alias address.
	`ioctl(fd, SIOCACLADDR6, (caddr_t)&ifra);`
	`struct in6_aliasreq ifra;`

ioctl command	Description

SIOCDCLADDR6	Delete an IPv6 cluster address.
	`ioctl(fd, SIOCDCLADDR6, (caddr_t)&ifr);`
	`struct in6_ifreq ifr;`

SIOCSIFADDRFLAG6	Set address source flag for an IPv6 address.
	`ioctl(fd, SIOCSIFADDRFLAG6, (caddr_t)&ifra2);`
	`struct in6_aliasreq2 ifra2;`

SIOCGIFADDRFLAG6	Get address source flag for an IPv6 address.
	`ioctl(fd, SIOCGIFADDRFLAG6, (caddr_t)&ifra2);`
	`struct in6_aliasreq2 ifra2;`

Return Values

Upon successful completion, ioctl returns `0`. Otherwise, it returns `-1` and sets `errno` to indicate the error.

Error Codes

The ioctl commands fail under the following general conditions:

Item	Description
EBADF	The file descriptor *fd* is not a valid open socket file descriptor.
EINTR	A signal was caught during ioctl operation.
EINVAL	An invalid command or argument was specified.

If the underlying operation specified by the ioctl command *cmd* failed, ioctl fails with one of the following error codes:

Item	Description
EACCES	Permission that is denied for the specified operation.
EADDRNOTAVAIL	Specified address not available for interface.
EAFNOSUPPORT	Operation that is not supported on sockets.
EBUSY	Resource is busy.
EEXIST	An entry or file exists.
EFAULT	Argument references an inaccessible memory area.
EIO	Input/Output error.
ENETUNREACH	Gateway unreachable.
ENOBUFS	Routing table overflow.
ENOCONNECT	No connection.
ENOMEM	Not enough memory available.
ENOTCONN	The operation is only defined on a connected socket, but the socket was not connected.
ENXIO	Device does not exist.
ESRCH	No such process.

Related information
Socket Overview
ioctl subroutine

isinet_addr Subroutine

Purpose

Determines if the given ASCII string contains an Internet address using dot notation.

Library

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

Syntax

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

u_long isinet_addr (name)
char * name;
```

Description

The **isinet_addr** subroutine determines if the given ASCII string contains an Internet address using dot notation (for example, "120.121.122.123"). The **isaddr_inet** subroutine considers Internet address strings as a valid string, and considers any other string type as an invalid strings.

The **isinet_addr** subrountine expects the ASCII string to conform to the following format:

```
string ::= field | field delimited_field^1-3
delimited_field ::= delimiter field
delimiter ::= .
field ::= 0 X | 0 x | 0 X hexadecimal* | 0 x hexadecimal* | decimal* | 0 octal*
hexadecimal ::= decimal | a | b | c | d | e | f | A | B | C | D | E | F
decimal ::= octal | 8 | 9
octal ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

Value	Description
A^*n*	Indicates *n* repetitions of pattern A.
A^*n*-*m*	Indicates *n* to *m* repetitions of pattern A.
A*	Indicates zero or more repetitions of pattern A, up to environmental limits.

The BNF description explicitly states the space character (' '), if used.

Value	Description
{*text*}	Indicates *text*, not a BNF symbol.

The **isinet_addr** subrountine allows the application to terminate the string with a null terminator (0x00) or a space (0x30). It ignores characters trailing the space character and considers the string invalid if the application does not terminate the string with a null terminator (0x00) or space (0x30).

The following describes the restrictions on the field values:

Address Format	Field Restrictions (values in decimal base)
a	a < 4294967296.
a.b	a < 256; b < 16777216.

Address Format	Field Restrictions (values in decimal base)
a.b.c	a < 256; b < 256; c < 16777216.
a.b.c.d	a < 256; b < 2^8; c < 256; d < 256.

The **isinet_addr** subrountine applications can enter field values exceeding the field value restrictions specified previously; **isinet_addr** accepts the least significant bits up to an integer in length. The **isinet_addr** subroutine still checks to see if the truncated value exceeds the maximum field value. For example, if an application gives the string `0.0;0;0xFF00000001` then **isinet_addr** interprets the string as `0.0.0.0x00000001` and considers the string as valid.

isinet_addr applications cannot omit field values between delimiters and considers a string with successive periods as invalid.

Examples of valid strings:

Input String	Comment
1	**isinet_addr** uses **a** format.
1.2	**isinet_addr** uses **a.b** format.
1.2.3.4	**isinet_add**r uses **a.b.c.d** format.
0x01.0X2.03.004	**isinet_addr** uses **a.b.c.d** format.
1.2 3.4	**isinet_addr** uses **a.b** format; and ignores "3.4".

Examples of invalid strings:

Input String	Reason
...	No explicit field values specified.
1.2.3.4.5	Excessive fields.
1.2.3.4.	Excessive delimiters and fields.
1,2	Bad delimiter.
1p	String not terminated by null terminator nor space.
{empty string}	No field or delimiter present.
9999.1.1.1	Value for field a exceeds limit.

Note:

1. The **isinet_addr** subroutine does not check the pointer to the ASCII string; the user takes responsibility for ensuring validity of the address of the ASCII string.
2. The application assumes responsibility for verifying that the network number and host number for the Internet address conforms to a class A or B or C Internet address; any other string is processed as a class C address.

All applications using **isinet_addr** must compile with the **_BSD** macro defined. Also, all socket applications must include the BSD library **libbsd** when applicable.

Parameters

Item	Description
name	Address of ASCII string buffer.

Return Values

The **isinet_addr** subroutine returns 1 for valid input strings and 0 for invalid input strings. **isinet_addr** returns the value as an unsigned long type.

Files

#include <ctype.h>

#include <sys/types.h>

kvalid_user Subroutine

Purpose

This routine maps the DCE principal to the local user account and determines if the DCE principal is allowed access to the account.

Library

Valid User Library (**libvaliduser.a**)

Syntax

Description

This routine is called when Kerberos 5 authentication is configured to determine if the incoming Kerberos 5 ticket should allow access to the local account.

This routine determines whether the DCE principal, specified by the *princ_name* parameter, is allowed access to the user's account identified by the *local_user* parameter. The routine accesses the **$HOME/.k5login** file for the users account. It looks for the string pointed to by *princ_name* in that file.

Access is granted if one of two things is true.

1. The **$HOME/.k5login** file exists and the *princ_name* is in it.
2. The **$HOME/.k5login** file does NOT exist and the DCE principal name is the same as the local user's name.

Parameters

Item	Description
princ_name	This parameter is a single-string representation of the Kerberos 5 principal. The Kerberos 5 libraries have two services, krb5_unparse_name and krb5_parse_name, which convert a krb5_principal structure to and from a single-string format. This routine expects the *princ_name* parameter to be a single-string form of the krb5_principal structure.
local_user	This parameter is the character string holding the name of the local account.

Return Values

If the user is allowed access to the account, the **kvalid_user** routine returns TRUE.

If the user is NOT allowed access to the account or there was an error, the **kvalid_user** routine returns FALSE.

Related information

Communications and networks

Authentication and the secure rcmds

listen Subroutine

Purpose

Listens for socket connections and limits the backlog of incoming connections.

Library

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

Syntax

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

int listen ( Socket,  Backlog)
int Socket, Backlog;
```

Description

The **listen** subroutine performs the following activities:

1. Identifies the socket that receives the connections.
2. Marks the socket as accepting connections.
3. Limits the number of outstanding connection requests in the system queue.

The outstanding connection request queue length limit is specified by the parameter *backlog* per listen call. A *no* parameter - *somaxconn* - defines the maximum queue length limit allowed on the system, so the effective queue length limit will be either *backlog* or *somaxconn*, whichever is smaller.

All applications containing the **listen** subroutine must be compiled with the **_BSD** macro 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
Socket	Specifies the unique name for the socket.
Backlog	Specifies the maximum number of outstanding connection requests.

Return Values

Upon successful completion, the **listen** subroutine returns a value 0.

If the **listen** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.

Error	Description
ECONNREFUSED	The host refused service, usually due to a server process missing at the requested name or the request exceeding the backlog amount.
EINVAL	The socket is already connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The referenced socket is not a type that supports the **listen** subroutine.

Examples

The following program fragment illustrates the use of the **listen** subroutine with 5 as the maximum number of outstanding connections which may be queued awaiting acceptance by the server process.

```
listen(s,5)
```

Related reference
accept Subroutine
Related information
Accepting Internet Stream Connections Example Program
Sockets Overview
Understanding Socket Connections

n

AIX runtime services beginning with the letter *n*.

ntohl Subroutine

Purpose

Converts an unsigned long integer from Internet network standard byte order to host byte order.

Library

ISODE Library (**libisode.a**)

Syntax

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

uint32_t ntohl (*NetLong*)
uint32_t *NetLong*;

Description

The **ntohl** subroutine converts an unsigned long (32-bit) integer from Internet network standard byte order to host byte order.

Receiving hosts require addresses and ports in host byte order. Use the **ntohl** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohl** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is same as the network byte order.

The **ntohl** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not same as the network byte order.

All applications containing the **ntohl** subroutine must be compiled with the **_BSD** macro 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
NetLong	Requires a 32-bit integer in network byte order.

Return Values

The **ntohl** subroutine returns a 32-bit integer in host byte order.

Related information
Sockets Overview

ntohll Subroutine

Purpose

Converts an unsigned long integer from Internet network standard byte order to host byte order.

Library

ISODE Library (**libisode.a**)

Syntax

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

uint64_t ntohll ( NetLong)
uint64_t NetLong;
```

Description

The **ntohll** subroutine converts an unsigned long (64-bit) integer from Internet network standard byte order to host byte order.

Receiving hosts require addresses and ports in host byte order. Use the **ntohll** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohll** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is the same as the network byte order.

The **ntohll** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not the same as the network byte order.

All applications containing the **ntohll** subroutine must be compiled with the **_BSD** macro 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
NetLong	Requires a 64-bit integer in network byte order.

Return Values

The **ntohll** subroutine returns a 64-bit integer in host byte order.

Related information
Sockets Overview

ntohs Subroutine

Purpose

Converts an unsigned short integer from Internet network byte order to host byte order.

Library

ISODE Library (**libisode.a**)

Syntax

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

uint16_t ntohs (*NetShort*)
uint16_t *NetShort*;

Description

The **ntohs** subroutine converts an unsigned short (16-bit) integer from Internet network byte order to the host byte order.

Receiving hosts require Internet addresses and ports in host byte order. Use the **ntohs** subroutine to convert Internet addresses and ports to the host integer representation.

The **ntohs** subroutine is defined in the **net/nh.h** file as a null macro if the host byte order is same as the network byte order.

The **ntohs** subroutine is declared in the **net/nh.h** file as a function if the host byte order is not same as the network byte order.

All applications containing the **ntohs** subroutine must be compiled with the **_BSD** macro 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
NetShort	Requires a 16-bit integer in network standard byte order.

Return Values

The **ntohs** subroutine returns a 16-bit integer in host byte order.

Related information
Sockets Overview

PostQueuedCompletionStatus Subroutine

Purpose

Post a completion packet to a specified I/O completion port.

Syntax

```
#include <iocp.h>
boolean_t PostQueuedCompletionStatus (CompletionPort, TransferCount, CompletionKey, Overlapped,
)
HANDLE CompletionPort;
DWORD TransferCount, CompletionKey;
LPOVERLAPPED Overlapped;
```

Description

The **PostQueuedCompletionStatus** subroutine attempts to post a completion packet to *CompletionPort* with the values of the completion packet populated by the *TransferCount*, *CompletionKey*, and *Overlapped* parameters.

The **PostQueuedCompletionStatus** subroutine returns a boolean indicating whether or not a completion packet has been posted.

The **PostQueuedCompletionStatus** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note: This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

Parameters

Item	Description
CompletionPort	Specifies the completion port that this subroutine will attempt to access.
TransferCount	Specifies the number of bytes transferred.
CompletionKey	Specifies the completion key.
Overlapped	Specifies the overlapped structure.

Return Values

Upon successful completion, the **PostQueuedCompletionStatus** subroutine returns a boolean indicating its success.

If the **PostQueuedCompletionStatus** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

Error Codes

The subroutine is unsuccessful if either of the following errors occur:

Item	Description
EBADF	The *CompletionPort* parameter was NULL.
EINVAL	The *CompletionPort* parameter was invalid.

Examples

The following program fragment illustrates the use of the **PostQueuedCompletionStatus** subroutine to post a completion packet.

```
c = GetQueuedCompletionStatus (34, 128, 25, struct overlapped);
```

Related information

Error Notification Object Class

r

AIX runtime services beginning with the letter *r*.

rcmd Subroutine

Purpose

Allows execution of commands on a remote host.

Library

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

Syntax

```
int rcmd (Host,
Port, LocalUser, RemoteUser, Command, ErrFileDesc)
char ** Host;
u_short  Port;
char * LocalUser;
char * RemoteUser;
char * Command;
int * ErrFileDesc;
```

Description

The **rcmd** subroutine allows execution of certain commands on a remote host that supports **rshd**, **rlogin**, and **rpc** among others.

Only processes with an effective user ID of root user can use the **rcmd** subroutine. An authentication scheme based on remote port numbers is used to verify permissions. Ports in the range between 0 and 1023 can only be used by a root user. The application must pass in *Port*, which must be in the range 512 to 1023.

The **rcmd** subroutine looks up a host by way of the name server or if the local name server isn't running, in the **/etc/hosts** file.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and given to the remote command as standard input (**stdin**) and standard output (**stdout**).

Always specify the *Host* parameter. If the local domain and remote domain are the same, specifying the domain parts is optional.

All applications containing the **rcmd** subroutine must be compiled with the **_BSD** macro 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
Host	Specifies the name of a remote host that is listed in the **/etc/hosts** file. If the specified name of the host is not found in this file, the **rcmd** subroutine is unsuccessful.
Port	Specifies the well-known port to use for the connection. The **/etc/services** file contains the DARPA Internet services, their ports, and socket types.
LocalUser and *RemoteUser*	Points to user names that are valid at the local and remote host, respectively. Any valid user name can be given.
Command	Specifies the name of the command to be started at the remote host.
ErrFileDesc	Specifies an integer controlling the set up of communication channels. Integer options are as follows: **Non-zero** Indicates an auxiliary channel to a control process is set up, and the *ErrFileDesc* parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. **0** Indicates the standard error (**stderr**) of the remote command is the same as standard output (**stdout**). No provision is made for sending arbitrary signals to the remote process. However, it is possible to send out-of-band data to the remote command.

Return Values

Upon successful completion, the **rcmd** subroutine returns a valid socket descriptor.

Upon unsuccessful completion, the **rcmd** subroutine returns a value of -1. The subroutine returns a -1, if the effective user ID of the calling process is not root user or if the subroutine is unsuccessful to resolve the host.

Files

Item	Description
/etc/services	Contains the service names, ports, and socket type.
/etc/hosts	Contains host names and their addresses for hosts in a network.
/etc/resolv.conf	Contains the name server and domain name.

Related information

Sockets Overview

rcmd_af Subroutine

Purpose

Allows execution of commands on a remote host.

Syntax

```
int rcmd_af(char **ahost, unsigned short rport,
                  const char *locuser, const char *remuser,
                  const char *cmd, int *fd2p, int af)
```

Description

The **rcmd_af** subroutine allows execution of certain commands on a remote host that supports **rshd**, **rlogin**, and **rpc** among others. It behaves the same as the existing **rcmd()** function, but instead of creating only an AF_INET TCP socket, it can also create an AF_INET6 TCP socket. The existing **rcmd()** function cannot transparently use AF_INET6 sockets because an application would not be prepared to handle AF_INET6 addresses returned by subroutines such as **getpeername()** on the file descriptor created by **rcmd()**.

Only processes with an effective user ID of root user can use the **rcmd_af** subroutine. An authentication scheme based on remote port numbers is used to verify permissions. Ports in the range between 0 and 1023 can only be used by a root user.

The **rcmd_af** subroutine looks up a host by way of the name server or if the local name server is not running, in the **/etc/hosts** file.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and given to the remote command as standard input (**stdin**) and standard output (**stdout**).

Always specify the *ahost* parameter. If the local domain and remote domain are the same, specifying the domain parts is optional.

Parameters

Item	Description
ahost	Specifies the name of a remote host that is listed in the **/etc/hosts** file. If the specified name of the host is not found in this file, the **rcmd_af** subroutine is unsuccessful.
rport	Specifies the well-known port to use for the connection. The **/etc/services** file contains the DARPA Internet services, their ports, and socket types.
locuser	Points to user names that are valid at the local host. Any valid user name can be given.
remuser	Points to user names that are valid at the remote host. Any valid user name can be given.
cmd	Specifies the name of the command to be started at the remote host.

Item	Description
fd2p	Specifies an integer controlling the set up of communication channels. Integer options are as follows: **Non-zero** Indicates an auxiliary channel to a control process is set up, and the *fd2p* parameter points to the file descriptor for the channel. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. **0** Indicates the standard error (**stderr**) of the remote command is the same as standard output (**stdout**). No provision is made for sending arbitrary signals to the remote process. However, it is possible to send out-of-band data to the remote command.
af	The family argument is AF_INET, AF_INET6, or AF_UNSPEC. When either AF_INET or AF_INET6 is specified, this function will create a socket of the specified address family. When AF_UNSPEC is specified, it will try all possible address families until a connection can be established, and will return the associated socket of the connection.

Return Values

Upon successful completion, the **rcmd_af** subroutine returns a valid socket descriptor. Upon unsuccessful completion, the **rcmd_af** subroutine returns a value of –1. The subroutine returns a –1 if the effective user ID of the calling process is not the root user or if the subroutine is unsuccessful to resolve the host.

Files

Item	Description
/etc/services	Contains the service names, ports, and socket type.
/etc/hosts	Contains host names and their addresses for hosts in a network.
/etc/resolv.conf	Contains the name server and domain name.

rds Subroutine

Purpose

Reliable Datagram Sockets (RDS) provides reliable, in-order datagram delivery between sockets across various network transport.

Library

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/bypass.h>
#include <net/rds_rdma.h>
```

Description

RDS is an implementation of the RDS Application Programming Interface (API). RDS can be transported through InfiniBand and loopback. RDS through TCP is disabled. RDS uses the standard AF_INET addresses to identify the endpoints.

Socket Creation

RDS sockets are created as follows:

```
rds_socket = socket(AF_BYPASS, SOCK_SEQPACKET, BYPASSPROTO_RDS);
```

Socket Options

RDS supports multiple socket options through the **setsockopt** and **getsockopt** calls. The following options with the SOL_SOCKET socket level are important.

SO_RCVBUF
Specifies the size of the receive buffer. See Congestion Control.

SO_SNDBUF
Specifies the size of the send buffer. See Message Transmission.

SO_SNDTIMEO
Specifies the send timeout of the socket when you enqueue a message on a socket with a full queue in the blocking mode.

RDS also supports multiple protocol-specific options with the SOL_RDS socket level .

Binding

A new RDS has no local address when it is initially returned from the **socket** call. The socket must be bound to a local address by running the **bind** system call before any messages are sent or received. The **bind** call attaches the socket to a specific network transport, which is based on the type of interface the local address is attached to. From the point the call is attached to the socket, the socket can reach the destinations that are available through this network transport.

For instance, when binding to the address of an InfiniBand interface, such as ib0, the socket uses the InfiniBand transport system. If RDS is not able to associate a transport system with the specific address, it returns the EADDRNOTAVAIL value.

An RDS socket can only be bound to one address and only one socket can be bound to a specific address or port pair. If no port is specified in the binding address, an unbound port is selected at random.

RDS does not permit the application to bind a previously bound socket to another address. Binding to the INADDR_ANY wildcard address is not allowed.

Connecting

In the default mode of operation RDS uses unconnected sockets, and specifies destination address as an argument to the **sendmsg** subroutine. However, RDS allows sockets to be connected to a remote end point by using the **connect** subroutine. If a socket is connected, you can call the **sendmsg** subroutine without specifying a destination address and the subroutine uses the remote address that was previously provided.

Congestion Control

RDS does not have an explicit congestion control mechanism like the common streaming protocols such as TCP. The sockets have two queue limits that are the `send queue size` and the `receive queue size`. Messages are accounted based on the number of bytes of payload.

The `send queue size` limits the data that the local processes can queue on a local socket. If the limit exceeds, the kernel does not accept messages until the queue is free and messages are delivered and acknowledged by the remote host.

The `receive queue size` limits the data that RDS stores on the receive queue of a socket before marking the socket as **congested**. When a socket becomes congested, RDS sends a *congestion map* update to the other participating hosts, which are then expected to stop sending more messages to this port.

There is a timing window during which a remote host can continue to send messages to a congested port. RDS resolves the timing window by accepting messages even when the receive queue of the socket exceeds the limit.

When the application receives incoming messages from the receive queue by using the **recvmsg** system call, the number of bytes on the receive queue reduces below the receive queue size and the port is marked as `uncongested`. A congestion update is sent to all the participating hosts.

The values for the send buffer size and receive buffer size can be tuned by the application through the SO_SNDBUF and SO_RCVBUF socket options.

Blocking Behavior

The **sendmsg** and **recvmsg** calls can be blocked in various situations. A call can be blocked or returned with an error depending on the non-blocking setting of the file descriptor and the `MSG_DONTWAIT` message flag. If the file descriptor is set to blocking mode (which is the default), and the `MSG_DONTWAIT` flag is not specified, the call is blocked.

The **SO_SNDTIMEO** and **SO_RCVTIMEO** socket options are used to specify a timeout (in seconds) after which the call ends and returns an error. The default timeout is 0, which allows RDS to block indefinitely.

Message Transmission

Messages can be sent by using the **sendmsg** call after the RDS socket is bound. Message length cannot exceed 4 GB as the wire protocol uses an unsigned 32-bit integer to express the message length.

RDS does not support data that is out-of-band. Applications can send data to unicast addresses only, where broadcast or multicast are not supported.

A successful **sendmsg** call places the message in the transmit queue of the socket where it remains until the destination acknowledges that the message is no longer in the network or the application removes the message from the send queue.

Messages can be removed from the send queue with the **RDS_CANCEL_SENT_TO** socket option.

When a message is in the transmit queue, its payload bytes are considered. If an attempt is made to send a message when the transmit queue is not free, the call blocks or returns the `EAGAIN` value.

When messages are sent to a destination that is marked as congested, the call is blocked or the`ENOBUFS` value is returned.

A message that is sent with no payload bytes does not require any space in the send buffer of the destination but a message receipt is sent to the destination. The receiver cannot get any payload data but the address of the sender can be viewed.

Messages sent to a port to which no socket is bound is discarded by the destination host. No error messages are reported to the sender.

Message Receipt

Messages can be received with the **recvmsg** call on RDS after it is bound to a source address. RDS returns messages in the same order that the sender sent the messages.

The address of the sender is returned in the sockaddr_in structure pointed by the msg_name field, if the field is set.

If the MSG_PEEK flag is set, the first message on the receive queue is returned without removing the message from the queue.

The memory that is used by messages waiting to be delivered does not limit the number of messages that can be queued to be received. RDS attempts to control congestion.

If the length of the message exceeds the size of the buffer that is provided to **recvmsg** call, then the remaining bytes in the message are discarded and the MSG_TRUNC flag is set in the msg_flags field. In this case the **recvmsg** call, returns the number of bytes copied. It does not return the length of the entire message. If MSG_TRUNC is set in the flags argument to **recvmsg**, it returns the number of bytes in the entire message. You can view the size of the next message in the receive queue without providing a zero length buffer and setting the MSG_PEEK and MSG_TRUNC options in the flags argument.

The sending address of a zero-length message is provided in the msg_name field.

Control Messages

RDS uses control messages that is the ancillary data by using the **msg_control** and **msg_controllen** fields in the **sendmsg** and **recvmsg** calls. Control messages that are generated by RDS have a **cmsg_level** value of sol_rds. Most control messages are related to the zerocopy interface added in RDS version 3, and are described in the **rds-rdma** subroutine.

The only exception is the RDS_CMSG_CONG_UPDATE message.

Polling

Support for the **poll** interface is limited. POLLIN is returned when there is an RDS message, or a control message waiting in the receive queue of the socket. POLLOUT is returned when there is space on the send queue of the socket.

Sending messages to the congested ports requires special handling mechanism. When an application tries to send message to a congested destination, the system call returns the ENOBUFS value. RDS cannot poll for POLLOUT because the transmit queue can still accommodate the messages and the call to the **poll** interface might return immediately, even though the destination is congested.

You can perform one of the method to handle the congestion:

- Poll for the POLLIN option. By default, a process sleeping in the **poll** interface is activated when the congestion map is updated. The application can retry any previously congested send operation.
- Monitor the explicit congestion, which gives the application greater control.

With explicit monitoring, the application polls for POLLIN option as before, and additionally uses the **RDS_CONG_MONITOR** socket option to install a 64-bit mask value in the socket, where each bit corresponds to a group of ports. When a congestion update is received, RDS socket checks the set of ports that became uncongested against the bit mask that is installed in the socket. If they overlap, a control message is enqueued on the socket, and the application is activated. When **recvmsg** call is called, RDS gives the control message that contains the bitmap on the socket.

The congestion monitor bitmask can be set and queried by using the **setsockopt** call with the **RDS_CONG_MONITOR** option, and a pointer to the 64-bit mask variable.

Congestion updates are delivered to the application through the **RDS_CMSG_CONG_UPDATE** control messages. The control messages are delivered separately, but never with RDS data message. The cmsg_data field of the control message is an eight byte data that contains the 64-bit mask value.

Applications can use the following macros to test for and set bits in the bitmask:

```
#define RDS_CONG_MONITOR_SIZE   64
#define RDS_CONG_MONITOR_BIT(port)  (((unsigned int) port) % RDS_CONG_MONITOR_SIZE)
#define RDS_CONG_MONITOR_MASK(port) (1 << RDS_CONG_MONITOR_BIT(port))
```

Canceling Messages

An application can cancel messages from the send queue by using the **RDS_CANCEL_SENT_TO** socket option with the **setsockopt** call. The **setsockopt** call uses an optional `sockaddr_in` address structure as an argument. Only messages to the destination address that is specified by the `sockaddr_in` address are discarded. If no address is provided, all pending messages are discarded.

Note: This call affects messages that are not transmitted and messages that are transmitted but no acknowledgment is received from the remote host.

Reliability

If the **sendmsg** succeeds, RDS guarantees that the message is visible to **recvmsg** on a socket that is bound to the destination address as long as that destination socket remains open.

If there is no socket bound on the destination, the message is dropped. If the RDS that is sending messages is not sure that a socket is bound, it tries to send the message indefinitely until it is sure or the sent message is canceled.

If a socket is closed, the pending sent messages on the socket are canceled and can or cannot be seen by the receiver.

The **RDS_CANCEL_SENT_TO** socket option can be used to cancel all the pending messages to a given destination.

If a receiving socket is closed with pending messages, then the sender considers those messages as having left the network and will not retransmit them.

A message is seen by the **recvmsg** call unless the `MSG_PEEK` is specified. When the message is delivered it is removed from the transmit queue of the sending socket.

All messages sent from the same socket to the same destination is delivered in the order they are sent. Messages sent from different sockets, or to different destinations, are delivered randomly.

rds-info Subroutine

Purpose

Displays information from the kernel extension of the Reliable Datagram Sockets (RDS) .

Syntax

```
rds-info [-v ] [ -cknrst]
```

Description

The **rds-info** utility displays various sources of information that the RDS kernel module maintains. When you run the **rds-info** utility without any optional arguments, the output has all the information. When you specify the optional arguments, the information that is associated with those options is displayed.

Parameters

Item	Descriptor
-c	Displays global counters. Each counter increments after the event occurs. You cannot reset the counters. The set of the supported counters can change with time. The list of output fields includes: **CounterName** The name of the counter. These names are derived from the kernel and can change based on the capability of the kernel extension. **Value** The number of times the counter increments after the kernel module is loaded.
-k	Displays all the RDS sockets in the system. There is one socket that is listed at a time that is not bound to or connected to any address because the **rds-info** utility uses an unbound socket to collect information. The list of output fields includes: **BoundAddr, BPort** The IP address and port number to which the socket is bound. The 0.0.0.0 0 address indicates that the socket is not bound. **ConnAddr, CPort** The IP address and port number to which the socket is connected. The 0.0.0.0 0 address indicates that the socket is not connected. **SndBuf, RcvBuf** The message payload in bytes that can be queued for sending or receiving on the respective socket.

Item	Descriptor
`-n`	Displays all the RDS connections. RDS connections are maintained between nodes by the network transports. The list of output fields includes: **LocalAddr** The IP address of a node. For connections that originate and terminate on the same node, the local address indicates the address that initiated the connection establishment and **RemoteAddr** The IP address of the remote end of the connection. **NextTX** The sequence number that is given to the next message that is sent over the connection. **NextRX** The expected sequence number of the next message that arrives over the connection. Any incoming messages with sequence numbers less than the expected number is dropped. **Flg** Flags that indicate the state of the connection. **s** A process is sending a message down the connection. **c** The transport is attempting to connect to the remote address. **C** The connection to the remote host is active.
`-r`, `-s`, `-t`	Displays the messages in the receive, send, or retransmit queues. **LocalAddr, LPort** The local IP address and port number of the node that is associated with the message. For sent messages, this address is the source address. For receive messages, this address is the destination address. **RemoteAddr, RPort** The remote IP address and port number that is associated with the message. For sent messages, this address is the destination address. For receive messages, this address is the source address. **Seq** The sequence number of the message. **Bytes** The message payload in bytes.
`-v`	Displays verbose output. When this option is specified complete data is displayed.

rds-ping Subroutine

Purpose

Tests the reachability of the remote node over Reliable Datagram Sockets (RDS).

Syntax

rds-ping [-c*count*][-i*interval*][-I*local_addr*]*remote_addr*

Description

The **rds-ping** utility is used to test whether a remote node is reachable over RDS. The RDS interface is designed to operate like the standard ping utility, with a difference. The **rds-ping** utility opens several RDS sockets and sends packets to port 0 on the specified host. This port is a special port number to which no socket is bound to, and the kernel processes the incoming packets and responds to them.

Parameters

Item	Description
-c *count*	Causes the **rds-ping** utility to exit after the specified number of packets are sent and received.
-I *address*	Accepts the local source address for the RDS socket that is based on the routing information for the specified destination address. For example, if packets to a specific destination are routed through the ib0 interface, it uses the IP address of ib0 as the source address. By using the -I option, you can override this choice.
-i *timeout*	Waits for one second between sending packets, by default. Use this option to specify a different interval. The timeout value is given in seconds, and can be a floating point number. Optionally, append the msec or usec parameter to specify the timeout in milliseconds or microseconds. **Note:** If you specify a timeout that is considerably smaller than the packet round-trip time, it produces unexpected results.

rds-rdma Subroutine

Purpose

Reliable Datagram Sockets (RDS) zerocopy provides an interface for remote direct memory access (RDMA) over RDS.

Description

The zerocopy interface of RDS was added in RDS Version3. In the RDS zerocopy, the client initiates a direct transfer to or from an area of the memory in its process address space. This memory need not be aligned.

The client obtains a handle for this region of memory, and passes it to the server. This cookie is called the RDMA cookie. To the application, the cookie is an opaque 64-bit data type.

The client sends this handle to the server application, along with other details of the RDMA request such as the data to transfer to the RDMA memory area. This message is called the RDMA request.

The server uses the RDMA cookie to initiate the requested RDMA transfer. The RDMA transfer is combined atomically with a normal RDS message, which is delivered to the client. This message is called the RDMA ACK. Atomic refers to both the RDMA succeeds and the RDMA ACK delivered, or they do not succeed.

When the client receives the RDMA ACK, it means that the RDMA completed successfully. If required, it can then release the RDMA cookie for this memory region.

RDMA operations are not reliable. Unlike normal RDS messages, RDS RDMA operations fail and get dropped.

Interface

The interface is based on control messages that are sent or received through the **sendmsg** and **recvmsg** system calls. Optionally, a previous interface can be used that is based on the **setsockopt** system call. The control messages must be used as it reduces the number of system calls required.

Control Message Interface

With the control message interface, the RDMA cookie is passed to the server out-of-band that is included in an extension header that is attached to the RDS message.

Initially, the client sends RDMA requests along with a RDS_CMSG_RDMA_MAP control message. The control message contains the address and length of the memory region to obtain a handle, flags, and a pointer to a memory location in the address space of the caller where the kernel stores the RDMA cookie.

If the application has an RDMA cookie for the memory range to or from an RDMA request, it can give this cookie to the kernel by using the RDS_CMSG_RDMA_DEST control message.

The kernel includes the resulting RDMA cookie in an extension header that is transmitted as part of the RDMA request to the server.

When the server receives the RDMA request, the kernel delivers the cookie within a RDS_CMSG_RDMA_DEST message. The server initiates the data transfer by sending the RDMA ACK message along with a RDS_CMSG_RDMA_ARGS control message. This message contains the RDMA cookie, and the local memory that can be copied.

The server process can request a notification when an RDMA operation completes. The notifications are delivered as the RDS_CMSG_RDMA_STATUS control messages. When an application calls the **recvmsg** call , it receives a regular RDS message with other RDMA-related control messages, or an empty message with one or more status control messages. When an RDMA operation fails and is discarded, the application can ask notifications for failed messages, regardless of the success notification of an individual message.

To activate the option for receiving failed notification, you must set the RDS_RECVERR socket option.

Setsockopt Interface

A process can register and release memory ranges for RDMA through the **setsockopt** calls with the help of RDS.

RDS_GET_MR
To obtain an RDMA cookie for a memory range, the application can use the **setsockopt** call with the RDS_GET_MR option. This cookie operates as the RDS_CMSG_RDMA_MAP control message. The argument contains the address and length of the memory range to be registered, and a pointer to an RDMA cookie variable where the system call stores the cookie for the registered range.

RDS_FREE_MR
Memory ranges are released by calling the **setsockopt** call with the RDS_FREE_MR option. You can specify the RDMA cookie with flags as arguments.

RDS_RECVERR
This is a Boolean option that is set and queried by using the **getsockopt** call. When enabled, RDS sends RDMA notification messages to the application for any RDMA operation that fails. This option by default is set to off.

For all the calls, the level argument to the **setsockopt** call is SOL_RDS.

RDMA Macros and types

RDMA cookie

```
typedef u_int64_t       rds_rdma_cookie_t
```

This cookie contains a memory location in the client process. The cookie contains the R_Key of the remote memory region, and the offset into it so that the alignment is not a concern for the application. The RDMA cookie is used in several struct types. The RDS_CMSG_RDMA_DEST control message contains a `rds_rdma_cookie_t` as payload.

Mapping arguments

The following data type is used with the RDS_CMSG_RDMA_MAP control messages and with the RDS_GET_MR socket option:

```
struct rds_iovec {
        u_int64_t       addr;
        u_int64_t       bytes;
};

struct rds_get_mr_args {
        struct rds_iovec vec;
        u_int64_t       cookie_addr;
        uint64_t        flags;
};
```

The `cookie_addr` parameter specifies a memory location to store the RDMA cookie.

The flags value is a bitwise OR of any of the following flags:

RDS_RDMA_USE_ONCE
This flag specifies to the kernel that the allocated RDMA cookie must be used one time. When the RDMA ACK message is received, the kernel automically unbinds the memory area and releases any resources that are associated with the cookie. If this flag is not set, the application must release the memory region by using the RDS_FREE_MR socket option.

RDS_RDMA_INVALIDATE
The RDMA memory mappings are not invalidated because it requires synchronization with the HCA, which is not cost effective. However, the server application can access the registered memory for any amount of time. The RDS code invalidates the mapping at the time it is released, and this can happen in two ways:

1. When an RDMA ACK and the RDS_RDMA_USE_ONCE flag is set
2. When the application releases the memory by using the RDS_FREE_MR socket option.

RDMA Operation

RDMA operations are initiated by the server by using the RDS_CMSG_RDMA_ARGS control message, which takes the following data as payload:

```
struct rds_rdma_args {
        rds_rdma_cookie_t cookie;
        struct rds_iovec remote_vec;
        u_int64_t       local_vec_addr;
        u_int64_t       nr_local;
        u_int64_t       flags;
        u_int32_t       user_token;
};
```

The cookie argument contains the RDMA cookie received from the client. The local memory has an array of `rds_iovecs`. The array address is specified in the `local_vec_addr` option, and its number of elements is specified in the `nr_local` option. The struct member remote_vec specifies a location relative to the memory area that is identified by the remote_vec.addr cookie as an offset into that region, and `remote_vec.bytes` is the length of the memory window that can be copied. This length must match the size of the local memory area that is the sum of bytes in all members of the local **iovec** call. The flags field contains the bitwise or the following flags:

RDS_RDMA_READWRITE
Performs an RDMA WRITE from the memory of the server to the client when the flag is set. If not set, RDS does an RDMA READ from the memory of the client to the memory of the server.

RDS_RDMA_FENCE
The order of an RDMA READ in reference to the subsequent SEND operations is not decided by InfiniBand. When this flag is set, the RDMA READ is separated from the subsequent RDS ACK message. Setting this flag requires an additional round trip of the InfiniBand. Set this flag by default.

RDS_RDMA_NOTIFY_ME
This flag requests a notification on completion of the RDMA operation whether successful or otherwise. The notification contains the value of the `user_token` field that is passed by the application. This flag allows the application to release resources such as buffers that are associated with the RDMA transfer. The `user_token` can be used to pass an application-specific identifier to the kernel. This token is returned to the application when a status notification is generated.

RDMA Notification

The RDS kernel code is able to notify the server application when an RDMA operation completes. These notifications are delivered through the RDS_CMSG_RDMA_STATUS control messages. By default, no notifications are generated. There are two ways an application can request for the messages. The status notifications can be enabled for every operation by setting the RDS_RDMA_NOTIFY_ME flag in the RDMA arguments. The application can request notifications for all RDMA operations that fail by setting the RDS_RECVERR socket option. In both cases, the format of the notification is the same and one notification is sent for the completed operation. The format of the message is as shown:

```
 struct rds_rdma_notify {
        u_int32_t       user_token;
        int32_t         status;
};
```

The `user_token` field contains the value that was previously stored in the kernel in the RDS_CMSG_RDMA_ARGS control message. The status field contains a status value, with 0 indicating success, and non-zero indicating an error. The following status codes are defined:

RDS_RDMA_SUCCESS
The RDMA operation succeeded.

RDS_RDMA_REMOTE_ERROR
The RDMA operation failed due to a remote access error. This error is because of an invalid R_key, offset, or transfer size.

RDS_RDMA_CANCELED
The RDMA operation was canceled by the application.

RDS_RDMA_DROPPED
RDMA operations was discarded after the connection failed and was reestablished. The RDMA operation is processed partially.

RDS_RDMA_OTHER_ERROR
Any other failure.

RDMA setsockopt arguments

When you use the RDS_GET_MR socket option to register a memory range, the application passes a pointer to a struct rds_get_mr_args variable. The **RDS_FREE_MR** call accepts an argument of type rds_free_mr_args struct:

```
struct rds_free_mr_args {
        rds_rdma_cookie_t cookie;
        u_int64_t       flags;
};
```

Where cookie specifies the RDMA cookie to be released. RDMA access to the memory range is not received instantly because the operation is costly. However, if the flags argument contains RDS_RDMA_INVALIDATE, RDS invalidates the mapping immediately. If the cookie argument is 0, and RDS_RDMA_INVALIDATE is set, RDS invalidates old memory mappings on all devices.

Errors

In addition to the usual error codes returned by **sendmsg**, **recvmsg** and **setsockopt** system calls, RDS returns the following error codes:

EAGAIN
RDS was unable to map a memory range because the limit exceeded (returned by RDS_CMSG_RDMA_MAP and RDS_GET_MR) .

EINVAL
When a message is sent, there were conflicting control messages (For example, two RDMA_MAP messages, or a RDMA_MAP and a RDMA_DEST message). In a RDS_CMSG_RDMA_MAP or RDS_GET_MR operation, the application that is specified by the memory range is greater than the maximum size supported. The size of the local memory specified in the rds_iovec call does not match the size of the remote memory range when an RDMA operation with the RDS_CMSG_RDMA_ARGS was set up.

EBUSY
RDS was unable to obtain a DMA mapping for the indicated memory.

ReadFile Subroutine

Purpose

Reads data from a socket.

Syntax

```
#include <iocp.h>
boolean_t ReadFile (FileDescriptor, Buffer, ReadCount, AmountRead, Overlapped)
HANDLE FileDescriptor;
LPVOID Buffer;
DWORD ReadCount;
LPDWORD AmountRead;
LPOVERLAPPED Overlapped;
```

Description

The **ReadFile** subroutine reads the number of bytes specified by the *ReadCount* parameter from the *FileDescriptor* parameter into the buffer indicated by the *Buffer* parameter. The number of bytes read is saved in the *AmountRead* parameter. The *Overlapped* parameter indicates whether or not the operation can be handled asynchronously.

The **ReadFile** subroutine returns a boolean (an integer) indicating whether or not the request has been completed.

The **ReadFile** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note: This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

Parameters

Item	Description
FileDescriptor	Specifies a valid file descriptor obtained from a call to the **socket** or **accept** subroutines.
Buffer	Specifies the buffer from which the data will be read.
ReadCount	Specifies the maximum number of bytes to read.
AmountRead	Specifies the number of bytes read. The parameter is set by the subroutine.

Item	Description
Overlapped	Specifies an overlapped structure indicating whether or not the request can be handled asynchronously.

Return Values

Upon successful completion, the **ReadFile** subroutine returns a boolean indicating the request has been completed.

If the **ReadFile** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

Error Codes

The subroutine is unsuccessful if any of the following errors occur:

Item	Description
EINPROGRESS	The read request can not be immediately satisfied and will be handled asynchronously. A completion packet will be sent to the associated completion port upon completion.
EAGAIN	The read request cannot be immediately satisfied and cannot be handled asynchronously.
EINVAL	The *FileDescriptor* parameter is invalid.

Examples

The following program fragment illustrates the use of the **ReadFile** subroutine to synchronously read data from a socket:

```
void buffer;
int amount_read;
b = ReadFile (34, &buffer, 128, &amount_read, NULL);
```

The following program fragment illustrates the use of the **ReadFile** subroutine to asynchronously read data from a socket:

```
void buffer;
int amount_read;
LPOVERLAPPED overlapped;
b = ReadFile (34, &buffer, 128, &amount_read, overlapped);
```

Note: The request will only be handled asynchronously if it cannot be immediately satisfied.

Related information

Error Notification Object Class

recv Subroutine

Purpose

Receives messages from connected sockets.

Library

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

Syntax

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

```
int recv (Socket,
Buffer, Length, Flags)
int  Socket;
void * Buffer;
size_t  Length;
int  Flags;
```

Description

The **recv** subroutine receives messages from a connected socket. The **recvfrom** and **recvmsg** subroutines receive messages from both connected and unconnected sockets. However, they are usually used for unconnected sockets only.

The **recv** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recv** subroutine waits for a message to arrive, unless the socket is nonblocking. If a socket is nonblocking, the system returns an error.

Use the **select** subroutine to determine when more data arrives.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the socket descriptor.
Buffer	Specifies an address where the message should be placed.
Length	Specifies the size of the *Buffer* parameter.
Flags	Points to a value controlling the message reception. The **/usr/include/sys/socket.h** file defines the *Flags* parameter. The argument to receive a call is formed by logically ORing one or more of the following values: **MSG_OOB** Processes out-of-band data. The significance of out-of-band data is protocol-dependent. **MSG_PEEK** Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to **recv()** or a similar function. **MSG_WAITALL** Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket.

Return Values

Upon successful completion, the **recv** subroutine returns the length of the message in bytes.

If the **recv** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Returns a 0 if the connection disconnects.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **recv** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
ECONNRESET	The remote peer forces the connection to be closed.
EFAULT	The data was directed to be received into a nonexistent or protected part of the process address space. The *Buffer* parameter is not valid.
EINTR	A signal interrupted the **recv** subroutine before any data was available.
EINVAL	The **MSG_OOB** flag is set and no out-of-band data is available.
ENOBUF	Insufficient resources are available in the system to perform the operation.
ENOTCONN	A receive is attempted on a **SOCK_STREAM** socket that is not connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	**MSG_OOB** flag is set for a **SOCK_DGRAM** socket, or **MSG_OOB** flag is set for any **AF_UNIX** socket.
ETIMEDOUT	The connection timed out during connection establishment, or there was a transmission timeout on an active connection.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Related reference
recvmsg Subroutine
recvfrom Subroutine
shutdown Subroutine
Related information
fgets subroutine
read subroutine
Sockets Overview

recvfrom Subroutine

Purpose

Receives messages from sockets.

Library

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

Syntax

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

ssize_t recvfrom
(Socket, Buffer, Length, Flags, From, FromLength)
int  Socket;
void * Buffer;
size_t  Length,
int  Flags;
struct sockaddr * From;
socklen_t * FromLength;
```

Description

The **recvfrom** subroutine allows an application program to receive messages from unconnected sockets. The **recvfrom** subroutine is normally applied to unconnected sockets as it includes parameters that allow the calling program to specify the source point of the data to be received.

To return the source address of the message, specify a nonnull value for the *From* parameter. The *FromLength* parameter is a value-result parameter, initialized to the size of the buffer associated with the *From* parameter. On return, the **recvfrom** subroutine modifies the *FromLength* parameter to indicate the actual size of the stored address. The **recvfrom** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recvfrom** subroutine waits for a message to arrive, unless the socket is nonblocking. If the socket is nonblocking, the system returns an error.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the socket descriptor.
Buffer	Specifies an address where the message should be placed.
Length	Specifies the size of the *Buffer* parameter.
Flags	Points to a value controlling the message reception. The argument to receive a call is formed by logically ORing one or more of the values shown in the following list: **MSG_OOB** Processes out-of-band data. The significance of out-of-band data is protocol-dependent. **MSG_PEEK** Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to **recv()** or a similar function. **MSG_WAITALL** Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket.
From	Points to a socket structure, filled in with the source's address.
FromLength	Specifies the length of the sender's or source's address.

Return Values

If the **recvfrom** subroutine is successful, the subroutine returns the length of the message in bytes.

If the call is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **recvfrom** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
ECONNRESET	The remote peer forces the connection to be closed.
EFAULT	The data was directed to be received into a nonexistent or protected part of the process address space. The buffer is not valid.
EINTR	The receive is interrupted by a signal delivery before any data is available.
EINVAL	The **MSG_OOB** flag is set but no out-of-band data is available.
ENOBUF	Insufficient resources are available in the system to perform the operation.
ENOPROTOOPT	The protocol is not 64-bit supported.
ENOTCONN	A receive is attempted on a **SOCK_STREAM** socket that is not connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	**MSG_OOB** flag is set for a **SOCK_DGRAM** socket, or **MSG_OOB** flag is set for any **AF_UNIX** socket.
ETIMEDOUT	The connection timed out during connection establishment, or there was a transmission timeout on an active connection.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Related reference
recv Subroutine
Related information
fgets subroutine
select subroutine
Sockets Overview
Understanding Socket Data Transfer

recvmsg Subroutine

Purpose

Receives a message from any socket.

Library

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

Syntax

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

```
int recvmsg ( Socket,  Message,  Flags)
int Socket;
struct msghdr Message [ ];
int Flags;
```

```
int recvmmsg ( Socket, MessageVec, Num_msg,  Flags, Timeout)
int Socket;
struct mmsghdr MessageVec [ ];
unsigned int Num_msg ;
int Flags;
struct timespec *Timeout
```

Description

The **recvmsg** subroutine receives messages from unconnected or connected sockets. The **recvmsg** subroutine returns the length of the message. If a message is too long to fit in the supplied buffer, excess bytes may be truncated depending on the type of socket that issued the message.

If no messages are available at the socket, the **recvmsg** subroutine waits for a message to arrive. If the socket is nonblocking and no messages are available, the **recvmsg** subroutine is unsuccessful.

Use the **select** subroutine to determine when more data arrives.

The **recvmsg** subroutine uses a **msghdr** structure to decrease the number of directly supplied parameters. The **msghdr** structure is defined in the**sys/socket.h** file. In BSD 4.3 Reno, the size and members of the **msghdr** structure have been modified. Applications wanting to start the old structure need to compile with **COMPAT_43** defined. The default behavior is that of BSD 4.4.

All applications containing the **recvmsg** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

The `recvmmsg` subroutine is an extension of the `recvmsg` subroutine that receives multiple messages from a socket to the caller socket. This subroutine has performance benefits for some applications. The `recvmmsg` subroutine supports timeout for wait during the receive operation.

The **sockfd** argument is the file descriptor of the socket from which data is received. The **msgvec** argument is a pointer to an array of `mmsghdr` structures. These arguments are defined in the `sys/socket.h file`.

On return from the `recvmmsg` subroutine, successive elements of the `msgvec` structure are updated to contain information about each received message. The *msg_len* field contains the size of the received message. The sub fields of the *msg_hdr* field are updated as described in the `recvmsg` subroutine. The return value of the `recvmsg` call indicates the number of elements of the `msgvec` field that are updated.

Parameters

Item	Description
Socket	Specifies the unique name of the socket.
Message	Points to the address of the **msghdr** structure, which contains both the address for the incoming message and the space for the sender address.
Flags	Permits the subroutine to exercise control over the reception of messages. The *Flags* parameter that is used to receive a call is formed by logically ORing one or more of the values which are shown in the following list: **MSG_OOB** Processes out-of-band data. The significance of out-of-band data is protocol-dependent. **MSG_PEEK** Peeks at incoming data. The data continues to be treated as unread and will be read by the next call to **recv()** or a similar function. **MSG_WAITALL** Requests that the function not return until the requested number of bytes have been read. The function can return fewer than the requested number of bytes only if a signal is caught, the connection is terminated, or an error is pending for the socket. **MSG_WAITFORONE** Turns on the MSG_DONTWAIT flag after the first message is received. The **/sys/socket.h** file contains the possible values for the *Flags* parameter.

Item	Description
MessageVec	Points to an array of `mmsghdr` structures, which contain `msghdr` structures for incoming messages, space for the sender address and a value that represents the total number of elements in the array.
Num_msg	Defines the number of messages to receive before the control is returned to the calling socket.
Timeout	The *timeout* argument points to a `timespec` structure that defines a timeout value(specified in seconds plus nanoseconds) for the receive operation. If the timeout value is `NULL` a call to the `recvmmsg` subroutine is blocked until the `vlen` messages are received or until the timeout value expires. A nonblocking call to the `recvmmsg` subroutine reads all messages that are available (the limit is specified by the `vlen` parameter) at the sender socket and returns from the subroutine to the calling function immediately.

Return Values

Upon successful completion of `recvmsg` subroutine, the length of the message in bytes is returned and for the `recvmmsg` subroutine, the number of received messages is returned.

If the `recvmsg` or the `recvmmsg` subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **recvmsg** subroutine is unsuccessful if any of the following error codes occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
ECONNRESET	The remote peer forces the connection to be closed.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINTR	The **recvmsg** subroutine was interrupted by delivery of a signal before any data was available for the receive.
EINVAL	The length of the **msghdr** structure is invalid, or the **MSG_OOB** flag is set and no out-of-band data is available.
EMSGSIZE	The *msg_iovlen* member of the **msghdr** structure pointed to by *Message* is less than or equal to 0, or is greater than **IOV_MAX**.
ENOBUF	Insufficient resources are available in the system to perform the operation.
ENOPROTOOPT	The protocol is not 64-bit supported.
ENOTCONN	A receive is attempted on a **SOCK_STREAM** socket that is not connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	**MSG_OOB** flag is set for a **SOCK_DGRAM** socket, or **MSG_OOB** flag is set for any **AF_UNIX** socket.
ETIMEDOUT	The connection timed out during connection establishment, or there was a transmission timeout on an active connection.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Related reference
recv Subroutine
Related information
no subroutine
select subroutine
Sockets Overview

res_init Subroutine

Purpose

Searches for a default domain name and Internet address.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void res_init ( )
```

Description

The **res_init** subroutine reads the **/etc/resolv.conf** file for the default domain name and the Internet address of the initial hosts running the name server.

Note: If the **/etc/resolv.conf** file does not exist, the **res_init** subroutine attempts name resolution using the local **/etc/hosts** file. If the system is not using a domain name server, the **/etc/resolv.conf** file should not exist. The **/etc/hosts** file should be present on the system even if the system is using a name server. In this instance, the file should contain the host IDs that the system requires to function even if the name server is not functioning.

The **res_init** subroutine is one of a set of subroutines that form the resolver, a set of functions that translate domain names to Internet addresses. All resolver subroutines use the **/usr/include/resolv.h** file, which defines the **_res** structure. The **res_init** subroutine stores domain name information in the **_res** structure. Three environment variables, **LOCALDOMAIN**, **RES_TIMEOUT**, and **RES_RETRY**, affect default values related to the **_res** structure.

All applications containing the **res_init** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

For more information on the **_res** structure, see "Understanding Domain Name Resolution" in *Communications Programming Concepts*.

Files

Item	Description
/etc/resolv.conf	Contains the name server and domain name.

Item	Description
/etc/hosts	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.

Related information
Sockets Overview
Understanding Domain Name Resolution

res_mkquery Subroutine

Purpose

Makes query messages for name servers.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

```
int res_mkquery (Operation, DomName, Class, Type, Data, DataLength)
int res_mkquery (Reserved, Buffer, BufferLength)
int  Operation;
char * DomName;
int  Class,  Type;
char * Data;
int  DataLength;
struct rrec * Reserved;
char * Buffer;
int  BufferLength;
```

Description

The **res_mkquery** subroutine creates packets for name servers in the Internet domain. The subroutine also creates a standard query message. The *Buffer* parameter determines the location of this message.

The **res_mkquery** subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the **_res** data structure. The **/usr/include/resolv.h** file contains the **_res** structure definition.

All applications containing the **res_mkquery** subroutine must be compiled with the **_BSD** macro 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
Operation	Specifies a query type. The usual type is **QUERY**, but the parameter can be set to any of the query types defined in the **arpa/nameser.h** file.

Item	Description
DomName	Points to the name of the domain. If the *DomName* parameter points to a single label and the **RES_DEFNAMES** structure is set, as it is by default, the subroutine appends the *DomName* parameter to the current domain name. The current domain name is defined by the name server in use or in the **/etc/resolv.conf** file.
Class	Specifies one of the following parameters: **C_IN** Specifies the ARPA Internet. **C_CHAOS** Specifies the Chaos network at MIT.
Type	Requires one of the following values: **T_A** Host address **T_NS** Authoritative server **T_MD** Mail destination **T_MF** Mail forwarder **T_CNAME** Canonical name **T_SOA** Start-of-authority zone **T_MB** Mailbox-domain name **T_MG** Mail-group member **T_MR** Mail-rename name **T_NULL** Null resource record **T_WKS** Well-known service **T_PTR** Domain name pointer **T_HINFO** Host information **T_MINFO** Mailbox information **T_MX** Mail-routing information **T_UINFO** User (**finger** command) information **T_UID** User ID **T_GID** Group ID

Item	Description
Data	Points to the data that is sent to the name server as a search key. The data is stored as a character array.
DataLength	Defines the size of the array pointed to by the *Data* parameter.
Reserved	Specifies a reserved and currently unused parameter.
Buffer	Points to a location containing the query message.
BufferLength	Specifies the length of the message pointed to by the *Buffer* parameter.

Return Values

Upon successful completion, the **res_mkquery** subroutine returns the size of the query. If the query is larger than the value of the *BufferLength* parameter, the subroutine is unsuccessful and returns a value of -1.

Files

Item	Description
/etc/resolv.conf	Contains the name server and domain name.

Related information
finger subroutine
Sockets Overview
Understanding Domain Name Resolution

res_ninit Subroutine

Purpose

Sets the default values for the members of the **_res** structure.

Library

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

Syntax

```
#include <resolv.h>

int res_ninit (statp)
res_state statp;
```

Description

Reads the **/etc/resolv.conf** configuration file to get the default domain name, search list, and internet address of the local name server(s). It does this in order to re-initialize the resolver context for a given thread in a multi-threaded environment.

The **res_ninit** subroutine sets the default values for the members of the **_res** structure (defined in the **/usr/include/resolv.h** file) after reading the **/etc/resolv.conf** configuration file to get default domain name, search list, Internet address of the local name server(s), sort list, and options (for details, please refer to the **/etc/resolv.conf** file). If no name server is configured, the server address is set to INADDR_ANY and the default domain name is obtained from the **gethostname** subroutine. It also

allows the user to override retrans, retry, and local domain definition using three environment variables RES_TIMEOUT, RES_RETRY, and LOCALDOMAIN, respectively.

Using this subroutine, each thread can have unique local resolver context. Since the configuration file is read each time the subroutine is called, it is capable of tracking dynamic changes to the resolver state file. Changes include, addition or removal of the configuration file or any other modifications to this file and reflect the same for a given thread. The **res_ninit** subroutine can also be used in single-threaded applications to detect dynamic changes to the resolver file even while the program is running (See the example section below). For more information on the **_res** structure, see Understanding Domain Name Resolution in *AIX Version 6.1 Communications Programming Concepts*.

Parameters

Item	Description
statp	Specifies the state to be initialized.

Examples

```
# cat /etc/resolv.conf
domain  in.ibm.com
nameserver      9.184.192.240
```

The following two examples use the **gethostbyname** system call to retrieve the host address of a system (florida.in.ibm.com) continuously. In the first example, **gethostbyname** is called (by a thread 'resolver') in a multi-threaded environment. The second example is not. Before each call to **gethostbyname**, the **res_ninit** subroutine is called to reflect dynamic changes to the configuration file.

```
1) #include <stdio.h>
    #include <netdb.h>
    #include <resolv.h>
    #include <pthread.h>

    void *resolver (void *arg);
    main( )  {
       pthread_t thid;
                   if ( pthread_create(&thid, NULL, resolver, NULL) ) {
                   printf("error in thread creation\n");
                   exit( );  }
                  pthread_exit(NULL);
     }

     void  *resolver (void *arg) {
            struct hostent *hp;
       struct sockaddr_in client;
           while(1) {
                   res_ninit(&_res);          /* res_init() with RES_INIT unset would NOT work
here */

                   hp = (struct hostent * ) gethostbyname("florida.in.ibm.com");
                   bcopy(hp->h_addr_list[0],&client.sin_addr,sizeof(client.sin_addr));
                   printf("hostname: %s\n",inet_ntoa(client.sin_addr));
             }
      }
```

If the **/etc/resolv.conf** file is present when the thread 'resolver' is invoked, the hostname will be resolved for that thread (using the nameserver 9.184.192.210) and the output will be `hostname: 9.182.21.151`.

If **/etc/resolv.conf** is not present, the output will be `hostname: 0.0.0.0`.

```
2) The changes to /etc/resolv.conf file are reflected even while the program is running

    #include <stdio.h>
    #include <resolv.h>
    #include <sys.h>
    #include <netdb.h>
    #include <string.h>
```

```
main() {
   struct hostent *hp;
       struct sockaddr_in client;

       while (1) {
              res_ninit(&_res);

              hp = (struct hostent * ) gethostbyname("florida.in.ibm.com");
              bcopy(hp->h_addr_list[0],&client.sin_addr,sizeof(client.sin_addr));
              printf("hostname: %s\n",inet_ntoa(client.sin_addr));
       }
 }
```

If **/etc/resolv.conf** is present while the program is running, the hostname will be resolved (using the nameserver 9.184.192.240) and the output will be `hostname: 9.182.21.151`.

If the **/etc/resolv.conf** file is not present, the output of the program will be `hostname: 0.0.0.0`.

Note: In the second example, the **res_init** subroutine with `_res.options = ~RES_INIT` can be used instead of the **res_ninit** subroutine.

Files

The **/etc/resolv.conf** and **/etc/hosts** files.

Related information

Understanding Domain Name Resolution

res_query Subroutine

Purpose

Provides an interface to the server query mechanism.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

```
int res_query (DomName, Class, Type, Answer, AnswerLength)
char * DomName;
int  Class;
int  Type;
u_char * Answer;
int  AnswerLength;
```

Description

The **res_query** subroutine provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified type and class for the fully-qualified domain name specified in the *DomName* parameter. The reply message is left in the answer buffer whose size is specified by the *AnswerLength* parameter, which is supplied by the caller.

The **res_query** subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. The **_res** data structure contains global information used by the resolver subroutines. The **/usr/include/resolv.h** file contains the **_res** structure definition.

All applications containing the **res_query** subroutine must be compiled with the **_BSD** macro 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
DomName	Points to the name of the domain. If the *DomName* parameter points to a single-component name and the **RES_DEFNAMES** structure is set, as it is by default, the subroutine appends the default domain name to the single-component name. The current domain name is defined by the name server in use or is specified in the **/etc/resolv.conf** file.
Class	Specifies one of the following values: **C_IN** Specifies the ARPA Internet. **C_CHAOS** Specifies the Chaos network at MIT.

Item	Description
Type	Requires one of the following values: **T_A** Host address **T_NS** Authoritative server **T_MD** Mail destination **T_MF** Mail forwarder **T_CNAME** Canonical name **T_SOA** Start-of-authority zone **T_MB** Mailbox-domain name **T_MG** Mail-group member **T_MR** Mail-rename name **T_NULL** Null resource record **T_WKS** Well-known service **T_PTR** Domain name pointer **T_HINFO** Host information **T_MINFO** Mailbox information **T_MX** Mail-routing information **T_UINFO** User (**finger** command) information **T_UID** User ID **T_GID** Group ID
Answer	Points to an address where the response is stored.
AnswerLength	Specifies the size of the answer buffer.

Return Values

Upon successful completion, the **res_query** subroutine returns the size of the response. Upon unsuccessful completion, the **res_query** subroutine returns a value of -1 and sets the **h_errno** value to the appropriate error.

Files

Item	Description
/etc/resolv.conf	Contains the name server and domain name.

Related information
finger subroutine
Sockets Overview
Understanding Domain Name Resolution

res_search Subroutine

Purpose

Makes a query and awaits a response.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

```
int res_search (DomName, Class, Type, Answer, AnswerLength)
char * DomName;
int  Class;
int  Type;
u_char * Answer;
int  AnswerLength;
```

Description

The **res_search** subroutine makes a query and awaits a response like the **res_query** subroutine. However, it also implements the default and search rules controlled by the **RES_DEFNAMES** and **RES_DNSRCH** options.

The **res_search** subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. The **_res** data structure contains global information used by the resolver subroutines. The **/usr/include/resolv.h** file contains the **_res** structure definition.

All applications containing the **res_search** subroutine must be compiled with the **_BSD** macro 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
DomName	Points to the name of the domain. If the *DomName* parameter points to a single-component name and the **RES_DEFNAMES** structure is set, as it is by default, the subroutine appends the default domain name to the single-component name. The current domain name is defined by the name server in use or is specified in the **/etc/resolv.conf** file. If the **RES_DNSRCH** bit is set, as it is by default, the **res_search** subroutine searches for host names in both the current domain and in parent domains.
Class	Specifies one of the following values: **C_IN** Specifies the ARPA Internet. **C_CHAOS** Specifies the Chaos network at MIT.

Item	Description
Type	Requires one of the following values: **T_A** Host address **T_NS** Authoritative server **T_MD** Mail destination **T_MF** Mail forwarder **T_CNAME** Canonical name **T_SOA** Start-of-authority zone **T_MB** Mailbox-domain name **T_MG** Mail-group member **T_MR** Mail-rename name **T_NULL** Null resource record **T_WKS** Well-known service **T_PTR** Domain name pointer **T_HINFO** Host information **T_MINFO** Mailbox information **T_MX** Mail-routing information **T_UINFO** User (**finger** command) information **T_UID** User ID **T_GID** Group ID
Answer	Points to an address where the response is stored.
AnswerLength	Specifies the size of the answer buffer.

Return Values

Upon successful completion, the **res_search** subroutine returns the size of the response. Upon unsuccessful completion, the **res_search** subroutine returns a value of -1 and sets the **h_errno** value to the appropriate error.

Files

Item	Description
/etc/resolv.conf	Contains the name server and domain name.

Related information
finger subroutine
Sockets Overview
Understanding Domain Name Resolution

res_send Subroutine

Purpose

Sends a query to a name server and retrieves a response.

Library

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

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>
```

```
int res_send (MessagePtr, MessageLength, Answer, AnswerLength)
char * MsgPtr;
int  MsgLength;
char * Answer;
int  AnswerLength;
```

Description

The **res_send** subroutine sends a query to name servers and calls the **res_init** subroutine if the **RES_INIT** option of the **_res** structure is not set. This subroutine sends the query to the local name server and handles time outs and retries.

The **res_send** subroutine is one of a set of subroutines that form the resolver, a set of functions that resolve domain names. Global information used by the resolver subroutines is kept in the **_res** structure. The **/usr/include/resolv.h** file contains the **_res** structure definition.

All applications containing the **res_send** subroutine must be compiled with the **_BSD** macro 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
MessagePtr	Points to the beginning of a message.
MessageLength	Specifies the length of the message.
Answer	Points to an address where the response is stored.
AnswerLength	Specifies the size of the answer area.

Return Values

Upon successful completion, the **res_send** subroutine returns the length of the message.

If the **res_send** subroutine is unsuccessful, the subroutine returns a -1.

Files

Item	Description
/etc/resolv.conf	Contains general name server and domain name information.

Related information
Sockets Overview
Understanding Domain Name Resolution

rexec Subroutine

Purpose

Allows command execution on a remote host.

Library

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

Syntax

```
int rexec ( Host, Port, User, Passwd, Command, ErrFileDescParam)
char **Host;
int Port;
char *User, *Passwd,
*Command;
int *ErrFileDescParam;
```

Description

The **rexec** subroutine allows the calling process to start commands on a remote host.

If the **rexec** connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and is given to the remote command as standard input and standard output.

All applications containing the **rexec** subroutine must be compiled with the **_BSD** macro 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
Host	Contains the name of a remote host that is listed in the **/etc/hosts** file or **/etc/resolv.config** file. If the name of the host is not found in either file, the **rexec** subroutine is unsuccessful.

Item	Description
Port	Specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call: `getservbyname("exec","tcp")`
User and *Passwd*	Points to a user ID and password valid at the host. If these parameters are not supplied, the **rexec** subroutine takes the following actions until finding a user ID and password to send to the remote host: 1. Searches the current environment for the user ID and password on the remote host. 2. Searches the user's home directory for a file called **$HOME/.netrc** that contains a user ID and password. 3. Prompts the user for a user ID and password.
Command	Points to the name of the command to be executed at the remote host.
ErrFileDescParam	Specifies one of the following values: **Non-zero** Indicates an auxiliary channel to a control process is set up, and a descriptor for it is placed in the *ErrFileDescParam* parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified. **0** Indicates the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out-of-band data to the remote command.

Return Values

Upon successful completion, the system returns a socket to the remote command.

If the **rexec** subroutine is unsuccessful, the system returns a -1 indicating that the specified host name does not exist.

Files

Item	Description
/etc/hosts	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.
/etc/resolv.conf	Contains the name server and domain name.
$HOME/.netrc	Contains automatic login information.

Related information

Transmission Control Protocol/Internet Protocol Sockets Overview

rexec_af Subroutine

Purpose

Allows command execution on a remote host.

Syntax

```
int rexec_af(char **ahost, unsigned short rport, const char *name,
                  const char *pass, const char *cmd, int *fd2p, int af)
```

Description

The **rexec_af** subroutine allows the calling process to start commands on a remote host. It behaves the same as the existing **rexec()** function, but instead of creating only an AF_INET TCP socket, it can also create an AF_INET6 TCP socket.

The **rexec_af** subroutine is useful because the existing **rexec()** function cannot transparently use AF_INET6 sockets. This is because an application would not be prepared to handle AF_INET6 addresses returned by functions such as **getpeername()** on the file descriptor created by **rexec()**.

If the **rexec_af** connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process and is given to the remote command as standard input and standard output.

All applications containing the **rexec_af** subroutine must be compiled with the **_BSD** macro 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
ahost	Contains the name of a remote host that is listed in the **/etc/hosts** file or **/etc/resolv.config** file. If the name of the host is not found in either file, the **rexec** subroutine is unsuccessful.
rport	Specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call: `getservbyname("exec","tcp")`
name and *pass*	Points to a valid user ID and password at the host. If these parameters are not supplied, the **rexec_af** subroutine takes the following actions until it finds a user ID and password to send to the remote host: 1. Searches the current environment for the user ID and password on the remote host. 2. Searches the user's home directory for a file called **$HOME/.netrc** that contains a user ID and password. 3. Prompts the user for a user ID and password.
cmd	Points to the name of the command to be executed at the remote host.

Item	Description
fd2p	Specifies one of the following values: **Non-zero** Indicates that an auxiliary channel to a control process is set up, and a descriptor for it is placed in the *fd2p* parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified. **0** Indicates that the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it might be possible to send out-of-band data to the remote command.
af	The family argument is AF_INET, AF_INET6, or AF_UNSPEC. When either AF_INET or AF_INET6 is specified, this subroutine will create a socket of the specified address family. When AF_UNSPEC is specified, it will try all possible address families until a connection can be established, and will return the associated socket of the connection.

Return Values

Upon successful completion, the system returns a socket to the remote command. If the **rexec_af** subroutine is unsuccessful, the system returns a –1, indicating that the specified host name does not exist.

Files

Item	Description
/etc/hosts	Contains host names and their addresses for hosts in a network. This file is used to resolve a host name into an Internet address.
/etc/resolv.conf	Contains the name server and domain name.
$HOME/.netrc	Contains automatic login information.

rresvport Subroutine

Purpose

Retrieves a socket with a privileged address.

Library

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

Syntax

```
int rresvport ( Port)
int *Port;
```

Description

The **rresvport** subroutine obtains a socket with a privileged address bound to the socket. A privileged Internet port is one that falls in a range between 0 and 1023.

Only processes with an effective user ID of root user can use the **rresvport** subroutine. An authentication scheme based on remote port numbers is used to verify permissions.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process.

All applications containing the **rresvport** subroutine must be compiled with the **_BSD** macro 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
Port	Specifies the port to use for the connection.

Return Values

Upon successful completion, the **rresvport** subroutine returns a valid, bound socket descriptor.

If the **rresvport** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **rresvport** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EAGAIN	All network ports are in use.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EMFILE	Two hundred file descriptors are currently open.
ENFILE	The system file table is full.
ENOBUFS	Insufficient buffers are available in the system to complete the subroutine.

Files

Item	Description
/etc/services	Contains the service names.

Related information
Sockets Overview

rresvport_af Subroutine

Purpose

Retrieves a socket with a privileged address.

Syntax

```
int  rresvport_af(int *port, int family);
```

Description

The **rresvport_af** subroutine obtains a socket with a privileged address bound to the socket. A privileged Internet port is one that falls in a range between 0 and 1023.

This subroutine is similar to the existing rresvport() subroutine, except that rresvport_af also takes and address family as an argument. This function is capable of creating either an AF_INET/TCP or an AF_INET6/TCP socket.

Only processes with an effective user ID of root user can use the **rresvport** subroutine. An authentication scheme based on remote port numbers is used to verify permissions.

If the connection succeeds, a socket in the Internet domain of type **SOCK_STREAM** is returned to the calling process.

All applications containing the **rresvport** subroutine must be compiled with the **_BSD** macro 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
port	Specifies the port to use for the connection.
family	Specifies either AF_INET or AF_INET6 to accommodate the appropriate version.

Return Values

Upon successful completion, the **rresvport_af** subroutine returns a valid, bound socket descriptor.

If the **rresvport_af** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

Item	Description
EAFNOSUPPORT	The address family is not supported.
EAGAIN	All network ports are in use.
EMFILE	Two hundred file descriptors are currently open.
ENFILE	The system file table is full.
ENOBUFS	Insufficient buffers are available in the system to complete the subroutine.

Files

Item	Description
/etc/services	Contains the service names.

ruserok Subroutine

Purpose

Allows servers to authenticate clients.

Library

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

Syntax

```
int ruserok (Host, RootUser, RemoteUser, LocalUser)
char * Host;
int  RootUser;
char * RemoteUser,
* LocalUser;
```

Description

The **ruserok** subroutine allows servers to authenticate clients requesting services.

Always specify the host name. If the local domain and remote domain are the same, specifying the domain parts is optional. To determine the domain of the host, use the **gethostname** subroutine.

All applications containing the **ruserok** subroutine must be compiled with the **_BSD** macro 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
Host	Specifies the name of a remote host. The **ruserok** subroutine checks for this host in the **/etc/host.equiv** file. Then, if necessary, the subroutine checks a file in the user's home directory at the server called **/$HOME/.rhosts** for a host and remote user ID.
RootUser	Specifies a value to indicate whether the effective user ID of the calling process is a root user. A value of 0 indicates the process does not have a root user ID. A value of 1 indicates that the process has local root user privileges, and the **/etc/hosts.equiv** file is not checked.
RemoteUser	Points to a user name that is valid at the remote host. Any valid user name can be specified.
LocalUser	Points to a user name that is valid at the local host. Any valid user name can be specified.

Return Values

The **ruserok** subroutine returns a 0, if the subroutine successfully locates the name specified by the *Host* parameter in the **/etc/hosts.equiv** file or the IDs specified by the *Host* and *RemoteUser* parameters are found in the **/$HOME/.rhosts** file.

If the name specified by the *Host* parameter was not found, the **ruserok** subroutine returns a -1.

Files

Item	Description
/etc/services	Contains service names.
/etc/host.equiv	Specifies foreign host names.
/$HOME/.rhosts	Specifies the remote users of a local user account.

Related information
Sockets Overview

S

AIX runtime services beginning with the letter *s*.

sctp_opt_info Subroutine

Purpose

Passes information both into and out of SCTP stack.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/sctp.h>

int sctp_opt_info(sd, id, opt, *arg_size, *size);
int sd;
sctp_assoc_t id;
int opt;
void *arg_size;
size_t *size;
```

Description

Applications use the **sctp_opt_info** subroutine to get information about various SCTP socket options from the stack. For the sockets with multiple associations, the association ID can be specified to apply the operation on any particular association of a socket. Because an SCTP association supports multihoming, this operation can be used to specify any particular peer address using a **sockaddr_storage** structure. In this case, the result of the operation will be applied to only that particular peer address.

Implementation Specifics

The **sctp_opt_info** subroutine is part of Base Operating System (BOS) Runtime.

Parameters

Item	Description
sd	Specifies the UDP style socket descriptor returned from the **socket** system call.
id	Specifies the identifier of the association to query.

Item	Description
opt	Specifies the socket option to get.
arg_size	Specifies an option specific structure buffer provided by the caller.
size	Specifies the size of the option returned.

Return Values

Upon successful completion, the **sctp_opt_info** subroutine returns 0.

If the **sctp_opt_info** subroutine is unsuccessful, the subroutine handler returns a value of -1 to the calling program and sets **errno** to the appropriate error code.

Error Codes

The **sctp_opt_info** subroutine is unsuccessful if any of the following errors occurs:

Item	Description
EFAULT	Indicates that the user has insufficient authority to access the data, or the address specified in the *uaddr* parameter is not valid.
EIO	Indicates that a permanent I/O error occurred while referencing data.
ENOMEM	Indicates insufficient memory for the required paging operation.
ENOSPC	Indicates insufficient file system or paging space.
ENOBUFS	Insufficient resources were available in the system to complete the call.
ENOPROTOOPT	Protocol not available.
ENOTSOCK	Indicates that the user has tried to do a socket operation on a non-socket.

Related information
Stream Control Transmission Protocol

sctp_peeloff Subroutine

Purpose

Branches off an association into a separate socket.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/sctp.h>

int sctp_peeloff(sd, *assoc_id);
int sd;
sctp_assoc_t *assoc_id;
```

Description

An application uses the **sctp_peeloff** subroutine when it wants to branch-off an existing association into a separate socket/file descriptor. It returns a new socket descriptor, which in turn can be used to send and receive subsequent SCTP packets. After it has been branched off, an association becomes completely independent of the original socket. Any subsequent data or control operations to that association must be passed using the new socket descriptor. Also, a close on the original socket descriptor will not close the new socket descriptor branched out of the association.

All the associations under the same socket share the same socket buffer space of the socket that they belong to. If an association gets branched off to a new socket using **sctp_peeloff**, then it inherits the socket buffer space associated with the new socket descriptor. This way, the association that got peeled off keeps more buffer space.

Implementation Specifics

The **sctp_peeloff** subroutine is part of Base Operating System (BOS) Runtime.

Parameters

Item	Description
sd	Specifies the UDP style socket descriptor returned from the **socket** system call.
assoc_id	Specifies the identifier of the association that is to be branched-off to a separate socket descriptor.

Return Values

Upon successful completion, the **sctp_peeloff** subroutine returns the nonnegative socket descriptor of the branched-off socket.

If the **sctp_peeloff** subroutine is unsuccessful, the subroutine handler returns a value of -1 to the calling program and moves an error code to the **errno** global variable.

Error Codes

The **sctp_peeloff** subroutine is unsuccessful if any of the following errors occurs:

Item	Description
EINVAL	Invalid argument.
EBADF	Bad file descriptor.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
ESOCKTNOSUPPORT	The socket in the specified address family is not supported.
EMFILE	The per-process descriptor table is full.
ENOBUFS	Insufficient resources were available in the system to complete the call.
ECONNABORTED	The client aborted the connection.

Related information

Stream Control Transmission Protocol

send Subroutine

Purpose

Sends messages from a connected socket.

Library

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

Syntax

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

```
int send (Socket,
Message, Length, Flags)
int  Socket;
const void * Message;
size_t  Length;
int  Flags;
```

Description

The **send** subroutine sends a message only when the socket is connected. This subroutine on a socket is not thread safe. The **sendto** and **sendmsg** subroutines can be used with unconnected or connected sockets.

To broadcast on a socket, first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

Specify the length of the message with the *Length* parameter. If the message is too long to pass through the underlying protocol, the system returns an error and does not transmit the message.

No indication of failure to deliver is implied in a **send** subroutine. A return value of -1 indicates some locally detected errors.

If no space for messages is available at the sending socket to hold the message to be transmitted, the **send** subroutine blocks unless the socket is in a nonblocking I/O mode. Use the **select** subroutine to determine when it is possible to send more data.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the unique name for the socket.
Message	Points to the address of the message to send.
Length	Specifies the length of the message in bytes.

Item	Description
Flags	Allows the sender to control the transmission of the message. The *Flags* parameter used to send a call is formed by logically ORing one or both of the values shown in the following list: **MSG_OOB** Processes out-of-band data on sockets that support **SOCK_STREAM** communication. **MSG_DONTROUTE** Sends without using routing tables. **MSG_MPEG2** Indicates that this block is a MPEG2 block. This flag is valid **SOCK_CONN_DGRAM** types of sockets only.

Return Values

Upon successful completion, the **send** subroutine returns the number of characters sent.

If the **send** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EACCES	Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.
EADDRNOTAVAIL	The specified address is not a valid address.
EAFNOSUPPORT	The specified address is not a valid address for the address family of this socket.
EBADF	The *Socket* parameter is not valid.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not in connection-mode and no peer address is set.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EHOSTUNREACH	The destination host cannot be reached.
EINTR	A signal interrupted **send** before any data was transmitted.
EINVAL	The *Length* parameter is invalid.
EISCONN	A **SOCK_DGRAM** socket is already connected.
EMSGSIZE	The message is too large to be sent all at once, as the socket requires.
ENETUNREACH	The destination network is not reachable.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
ENOENT	The path name does not name an existing file, or the path name is an empty string.
ENOMEM	The available data space in memory is not large enough to hold group/ACL information.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The socket argument is associated with a socket that does not support one or more of the values set in *Flags*.

Error	Description
EPIPE	An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. If the socket is of type **SOCK_STREAM**, the **SIGPIPE** signal is generated to the calling process.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Related reference
sendmsg Subroutine
setsockopt Subroutine
Related information
select subroutine
Sockets Overview

sendmsg Subroutine

Purpose

Sends a message from a socket using a message structure.

Library

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

Syntax

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

```
int sendmsg ( Socket,  Message,  Flags)
int Socket;
const struct msghdr Message [ ];
int Flags;
```

Description

The **sendmsg** subroutine sends messages through connected or unconnected sockets using the **msghdr** message structure. The **/usr/include/sys/socket.h** file contains the **msghdr** structure and defines the structure members. In BSD 4.4, the size and members of the **msghdr** message structure have been modified. Applications wanting to start the old structure need to compile with **COMPAT_43** defined. The default behaviour is that of BSD 4.4.

To broadcast on a socket, the application program must first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

The **sendmsg** subroutine supports only 15 message elements.

All applications containing the **sendmsg** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

The **sendmsg** routine supports IPv6 ancillary data elements as defined in the Advanced Sockets API for IPv6.

Parameters

Item	Description
Socket	Specifies the socket descriptor.
Message	Points to the **msghdr** message structure containing the message to be sent.
Flags	Allows the sender to control the message transmission. The **sys/socket.h** file contains the *Flags* parameter. The *Flags* parameter used to send a call is formed by logically ORing one or both of the following values: **MSG_OOB** Processes out-of-band data on sockets that support **SOCK_STREAM**. **Note:** The following value is not for general use. It is an administrative tool used for debugging or for routing programs. **MSG_DONTROUTE** Sends without using routing tables. **MSG_MPEG2** Indicates that this block is a MPEG2 block. It only applies to **SOCK_CONN_DGRAM** types of sockets only.

Return Values

Upon successful completion, the **sendmsg** subroutine returns the number of characters sent.

If the **sendmsg** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

The **sendmsg** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EACCES	Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.
EADDRNOTAVAIL	The specified address is not a valid address.
EAFNOSUPPORT	The specified address is not a valid address for the address family of this socket.
EBADF	The *Socket* parameter is not valid.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not in connection-mode and does not have its peer address set.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EHOSTUNREACH	The destination host cannot be reached.
EINTR	A signal interrupted **sendmsg** before any data was transmitted.
EINVAL	The length of the **msghdr** structure is invalid.
EISCONN	A **SOCK_DGRAM** socket is already connected.
EMSGSIZE	The message is too large to be sent all at once (as the socket requires), or the **msg_iovlen** member of the **msghdr** structure pointed to by *Message* is less than or equal to 0 or is greater than **IOV_MAX**.

Error	Description
ENOENT	The path name does not name an existing file, or the path name is an empty string.
ENETUNREACH	The destination network is not reachable.
ENOBUFS	The system ran out of memory for an internal data structure.
ENOMEM	The available data space in memory is not large enough to hold group/ACL information.
ENOPROTOOPT	The protocol is not 64-bit supported.
ENOTCONN	The socket is in connection-mode but is not connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The socket argument is associated with a socket that does not support one or more of the values set in flags.
EPIPE	An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. If the socket is of type **SOCK_STREAM**, the **SIGPIPE** signal is generated to the calling process.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Related reference
send Subroutine
setsockopt Subroutine
Related information
select subroutine
Sockets Overview

sendto Subroutine

Purpose

Sends messages through a socket.

Library

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

Syntax

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

int sendto
(Socket, Message, Length,
Flags, To, ToLength)
int  Socket;
const void * Message;
size_t  Length;
int  Flags;
const struct sockaddr * To;
socklen_t  ToLength;
```

Description

The **sendto** subroutine allows an application program to send messages through an unconnected socket by specifying a destination address.

To broadcast on a socket, first issue a **setsockopt** subroutine using the **SO_BROADCAST** option to gain broadcast permissions.

Provide the address of the target using the *To* parameter. Specify the length of the message with the *Length* parameter. If the message is too long to pass through the underlying protocol, the error **EMSGSIZE** is returned and the message is not transmitted.

If the **sending** socket has no space to hold the message to be transmitted, the **sendto** subroutine blocks the message unless the socket is in a nonblocking I/O mode.

Use the **select** subroutine to determine when it is possible to send more data.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the unique name for the socket.
Message	Specifies the address containing the message to be sent.
Length	Specifies the size of the message in bytes.
Flags	Allows the sender to control the message transmission. The *Flags* parameter used to send a call is formed by logically ORing one or both of the following values: **MSG_OOB** Processes out-of-band data on sockets that support **SOCK_STREAM**. **Note:** **MSG_DONTROUTE** Sends without using routing tables. The **/usr/include/sys/socket.h** file defines the *Flags* parameter.
To	Specifies the destination address for the message. The destination address is a **sockaddr** structure defined in the **/usr/include/sys/socket.h** file.
ToLength	Specifies the size of the destination address.

Return Values

Upon successful completion, the **sendto** subroutine returns the number of characters sent.

If the **sendto** subroutine is unsuccessful, the system returns a value of -1, and the **errno** global variable is set to indicate the error.

Error Codes

The subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EACCES	Write access to the named socket is denied, or the socket trying to send a broadcast packet does not have broadcast capability.
EADDRNOTAVAIL	The specified address is not a valid address.

Error	Description
EAFNOSUPPORT	The specified address is not a valid address for the address family of this socket.
EBADF	The *Socket* parameter is not valid.
ECONNRESET	A connection was forcibly closed by a peer.
EDESTADDRREQ	The socket is not in connection-mode and no peer address is set.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EHOSTUNREACH	The destination host cannot be reached.
EINTR	A signal interrupted **sendto** before any data was transmitted.
EINVAL	The *Length* or *ToLength* parameter is invalid.
EISCONN	A **SOCK_DGRAM** socket is already connected.
EMSGSIZE	The message is too large to be sent all at once as the socket requires.
ENETUNREACH	The destination network is not reachable.
ENOBUFS	The system ran out of memory for an internal data structure.
ENOENT	The path name does not name an existing file, or the path name is an empty string.
ENOMEM	The available data space in memory is not large enough to hold group/ACL information.
ENOPROTOOPT	The protocol is not 64-bit supported.
ENOTCONN	The socket is in connection-mode but is not connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EOPNOTSUPP	The socket argument is associated with a socket that does not support one or more of the values set in *Flags*.
EPIPE	An attempt was made to send on a socket that was connected, but the connection has been shut down either by the remote peer or by this side of the connection. If the socket is of type **SOCK_STREAM**, the **SIGPIPE** signal is generated to the calling process.
EWOULDBLOCK	The socket is marked nonblocking, and no connections are present to be accepted.

Related reference
setsockopt Subroutine
Related information
select subroutine
Sending Datagrams Example Program

send_file Subroutine

Purpose

Sends the contents of a file through a socket.

Library

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

Syntax

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

ssize_t send_file(Socket_p, sf_iobuf, flags)

int * Socket_p;
struct sf_parms * sf_iobuf;
uint_t  flags;
```

Description

The **send_file** subroutine sends data from the opened file specified in the *sf_iobuf* parameter, over the connected socket pointed to by the *Socket_p* parameter.

Note: Currently, the **send_file** only supports the TCP/IP protocol (SOCK_STREAM socket in AF_INET). An error will be returned when this function is used on any other types of sockets.

Parameters

Item	Description
Socket_p	Points to the socket descriptor of the socket which the file will be sent to. **Note:** This is different from most of the socket functions.

Item	Description
sf_iobuf	Points to a *sf_parms* structure defined as follows:

```
/*
 * Structure for the send_file system call
 */
#ifdef __64BIT__
#define SF_INT64(x)     int64_t x;
#define SF_UINT64(x)    uint64_t x;
#else
#ifdef _LONG_LONG
#define SF_INT64(x)     int64_t x;
#define SF_UINT64(x)    uint64_t x;
#else
#define SF_INT64(x)     int filler_##x; int x;
#define SF_UINT64(x)    int filler_##x; uint_t x;
#endif
#endif

struct sf_parms {
    /* --------- header parms ---------- */
    void      *header_data;         /* Input/Output. Points to header buf */
    uint_t      header_length;      /* Input/Output. Length of the header */
    /* --------- file parms ------------ */
    int       file_descriptor;      /* Input. File descriptor of the file */
    SF_UINT64(file_size)            /* Output. Size of the file */
    SF_UINT64(file_offset)          /* Input/Output. Starting offset */
    SF_INT64(file_bytes)            /* Input/Output. number of bytes to send */
    /* --------- trailer parms --------- */
   void      *trailer_data;         /* Input/Output. Points to trailer buf */
   uint_t    trailer_length;        /* Input/Output. Length of the trailer */
    /* --------- return info ----------- */
    SF_UINT64(bytes_sent)           /* Output. number of bytes sent */
};
```

header_data
Points to a buffer that contains header data which is to be sent before the file data. May be a NULL pointer if *header_length* is 0. This field will be updated by **send_file** when header is transmitted - that is, `header_data` + number of bytes of the header sent.

header_length
Specifies the number of bytes in the *header_data*. This field must be set to 0 to indicate that header data is not to be sent. This field will be updated by **send_file** when header is transmitted - that is, `header_length` - number of bytes of the header sent.

file_descriptor
Specifies the file descriptor for a file that has been opened and is readable. This is the descriptor for the file that contains the data to be transmitted. The *file_descriptor* is ignored when *file_bytes* = 0. This field is not updated by **send_file**.

file_size
Contains the byte size of the file specified by *file_descriptor*. This field is filled in by the kernel.

file_offset
Specifies the byte offset into the file from which to start sending data. This field is updated by the **send_file** when file data is transmitted - that is, `file_offset` + number of bytes of the file data sent.

Item	Description

file_bytes
Specifies the number of bytes from the file to be transmitted. Setting *file_bytes* to -1 transmits the entire file from the *file_offset*. When this field is not set to -1, it is updated by **send_file** when file data is transmitted - that is, `file_bytes` - number of bytes of the file data sent.

trailer_data
Points to a buffer that contains trailer data which is to be sent after the file data. May be a NULL pointer if *trailer_length* is 0. This field will be updated by **send_file** when trailer is transmitted - that is, `trailer_data` + number of bytes of the trailer sent.

trailer_length
Specifies the number of bytes in the *trailer_data*. This field must be set to 0 to indicate that trailer data is not to be sent. This field will be updated by **send_file** when trailer is transmitted - that is, `trailer_length` - number of bytes of the trailer sent.

bytes_sent
Contains number of bytes that were actually sent in this call to **send_file**. This field is filled in by the kernel.

All fields marked with `Input` in the *sf_parms* structure requires setup by an application prior to the **send_file** calls. All fields marked with `Output` in the *sf_parms* structure adjusts by **send_file** when it successfully transmitted data, that is, either the specified data transmission is partially or completely done.

The **send_file** subroutine attempts to write *header_length* bytes from the buffer pointed to by *header_data*, followed by *file_bytes* from the file associated with *file_descriptor*, followed by *trailer_length* bytes from the buffer pointed to by *trailer_data*, over the connection associated with the socket pointed to by *Socket_p*.

As the data is sent, the kernel updates the parameters pointed by *sf_iobuf* so that if the **send_file** has to be called multiple times (either due to interruptions by signals, or due to non-blocking I/O mode) in order to complete a file data transmission, the application can reissue the **send_file** command without setting or re-adjusting the parameters over and over again.

If the application sets *file_offset* greater than the actual file size, or *file_bytes* greater than (the actual file size - *file_offset*), the return value will be -1 with errno EINVAL.

Item	Description
flags	Specifies the following attributes: ***SF_CLOSE*** Closes the socket pointed to by *Socket_p* after the data has been successfully sent or queued for transmission. ***SF_REUSE*** Prepares the socket for reuse after the data has been successfully sent or queued for transmission and the existing connection closed. **Note:** This option is currently not supported on this operating system. ***SF_DONT_CACHE*** Does not put the specified file in the Network Buffer Cache. ***SF_SYNC_CACHE*** Verifies/Updates the Network Buffer Cache for the specified file before transmission. When the *SF_CLOSE* flag is set, the connected socket specified by *Socket_p* will be disconnected and closed by **send_file** after the requested transmission has been successfully done. The socket descriptor pointed to by *Socket_p* will be set to -1. This flag won't take effect if **send_file** returns non-0. The flag *SF_REUSE* currently is not supported by AIX. When this flag is specified, the socket pointed by *Socket_p* will be closed and returned as -1. A new socket needs to be created for the next connection. **send_file** will take advantage of a Network Buffer Cache in kernel memory to dynamically cache the output file data. This will help to improve the **send_file** performance for files which are: 1. accessed repetitively through network and 2. not changed frequently. Applications can exclude the specified file from being cached by using the *SF_DONT_CACHE* flag. **send_file** will update the cache every so often to make sure that the file data in cache is valid for a certain time period. The network option parameter "send_file_duration" controlled by the **no** command can be modified to configure the interval of the **send_file** cache validation, the default is 300 (in seconds). Applications can use the *SF_SYNC_CACHE* flag to ensure that a cache validation of the specified file will occur before the file is sent by **send_file**, regardless the value of the "send_file_duration". Other Network Buffer Cache related parameters are "nbc_limit", nbc_max_cache", and nbc_min_cache". For additional infromation, see the **no** command.

Return Value

There are three possible return values from **send_file**:

Value	Description
-1	an error has occurred, errno contains the error code.
0	the command has completed successfully.
1	the command was completed partially, some data has been transmitted but the command has to return for some reason, for example, the command was interrupted by signals.

The fields marked with `Output` in the *sf_parms* structure (pointed to by *sf_iobuf*) is updated by **send_file** when the return value is either 0 or 1. The *bytes_sent* field contains the total number of bytes that were sent in this call. It is always true that *bytes_sent* (`Output`) <= *header_length*(`Input`) + *file_bytes*(`Input`) + *trailer_length* (`Input`).

The **send_file** supports the blocking I/O mode and the non-blocking I/O mode. In the blocking I/O mode, **send_file** blocks until all file data (plus the header and the trailer) is sent. It adjusts the *sf_iobuf* to reflect the transmission results, and return 0. It is possible that **send_file** can be interrupted before the request is fully done, in that case, it adjusts the *sf_iobuf* to reflect the transmission progress, and return 1.

In the non-blocking I/O mode, the **send_file** transmits as much as the socket space allows, adjusts the *sf_iobuf* to reflect the transmission progress, and returns either 0 or 1. When there is no socket space in the system to buffer any of the data, the **send_file** returns -1 and sets errno to EWOULDBLOCK. **select** or **poll** can be used to determine when it is possible to send more data.

Possible errno returned:

EBADF	Either the socket or the file descriptor parameter is not valid.
ENOTSOCK	The socket parameter refers to a file, not a socket.
EPROTONOSUPPORT	Protocol not supported.
EFAULT	The addresses specified in the HeaderTailer parameter is not in a writable part of the user-address space.
EINTR	The operation was interrupted by a signal before any data was sent. (If some data was sent, **send_file** returns the number of bytes sent before the signal, and EINTR is not set).
EINVAL	The offset, length of the HeaderTrailer, or flags parameter is invalid.
ENOTCONN	A **send_file** on a socket that is not connected, a **send_file** on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.
ENOMEM	No memory is available in the system to perform the operation.

PerformanceNote

By taking advantage of the Network Buffer Cache, **send_file** provides better performance and network throughput for file transmission. It is recommanded for files bigger than 4K bytes.

Related information

select subroutine

Sockets Overview

Understanding Socket Data Transfer

set_auth_method Subroutine

Purpose

Sets the authentication methods for the rcmds for this system.

Library

Authentication Methods Library (**libauthm.a**)

Syntax

Description

This method configures the authentication methods for the system. The authentication methods should be passed to the function in the order in which they should be attempted in the unsigned integer pointer in which the user passed.

The list is an array of unsigned integers terminated by a zero. Each integer identifies an authentication method. The order that a client should attempt to authenticate is defined by the order of the list.

The flags identifying the authentication methods are defined in the **/usr/include/authm.h** file.

Any undefined bits in the input parameter invalidate the entire command. If the same authentication method is specified twice or if any authentication method is specified after Standard AIX, the command fails.

The user must have root authority or this method fails.

Parameter

Item	Description
authm	Points to an array of unsigned integers. The list of authentication methods to be set is terminated by a zero.

Return Values

Upon successful completion, the **set_auth_method** subroutine returns a zero.

Upon unsuccessful completion, the **set_auth_method** subroutine returns an **errno**.

Related information

Communications and networks

Authentication and the secure rcmds

setdomainname Subroutine

Purpose

Sets the name of the current domain.

Library

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

Syntax

```
int setdomainname ( Name,  Namelen)
char *Name;
int Namelen;
```

Description

The **setdomainname** subroutine sets the name of the domain for the host machine. It is normally used when the system is bootstrapped. You must have root user authority to run this subroutine.

The purpose of domains is to enable two distinct networks that may have host names in common to merge. Each network would be distinguished by having a different domain name. At the current time, only Network Information Service (NIS) makes use of domains set by this subroutine.

All applications containing the **setdomainname** subroutine must be compiled with the **_BSD** macro set to a specific value. Acceptable values are 43 and 44. In addition, all socket applications must include the BSD **libbsd.a** library.

Note: Domain names are restricted to 256 characters.

Parameters

Item	Description
Name	Specifies the domain name to be set.
Namelen	Specifies the size of the array pointed to by the *Name* parameter.

Return Values

If the call succeeds, a value of 0 is returned. If the call is unsuccessful, a value of -1 is returned and an error code is placed in the **errno** global variable.

Error Codes

The following errors may be returned by this subroutine:

Error	Description
EFAULT	The *Name* parameter gave an invalid address.
EPERM	The caller was not the root user.

Related reference
getdomainname Subroutine
Related information
Sockets Overview

sethostent Subroutine

Purpose

Opens network host file.

Library

```
Standard C Library (libc.a)
(libbind)
libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>

sethostent ( StayOpen)
int StayOpen;
```

Description

When using the **sethostent** subroutine in DNS/BIND name service resolution, **sethostent** allows a request for the use of a connected socket using TCP for queries. If the *StayOpen* parameter is non-zero, this sets

the option to send all queries to the name server using TCP and to retain the connection after each call to **gethostbyname** or **gethostbyaddr**.

When using the **sethostent subroutine** to search the **/etc/hosts** file, **sethostent** opens and rewinds the **/etc/hosts** file. If the *StayOpen* parameter is non-zero, the hosts database is not closed after each call to **gethostbyname** or **gethostbyaddr**.

Parameters

Item	Description
StayOpen	When used in NIS name resolution and to search the local **/etc/hosts** file, it contains a value used to indicate whether to close the host file after each call to **gethostbyname** and **gethostbyaddr**. A non-zero value indicates not to close the host file after each call and a zero value allows the file to be closed. When used in DNS/BIND name resolution, a non-zero value retains the TCP connection after each call to **gethostbyname** and **gethostbyaddr** . A value of zero allows the connection to be closed.

Files

Item	Description
/etc/hosts	Contains the host name database.
/etc/netsvc.conf	Contains the name services ordering.
/etc/include/netdb.h	Contains the network database structure.

Related reference
endhostent Subroutine
Related information
Sockets Overview
Network Address Translation

sethostent_r Subroutine

Purpose

Opens network host file.

Library

```
Standard C Library (libc.a)
(libbind)
libnis)
(liblocal)
```

Syntax

```
#include <netdb.h>
sethostent_r (StayOpenflag, ht_data)

int StayOpenflag;
struct hostent_data *ht_data;
```

Description

When using the **sethostent_r** subroutine in DNS/BIND name service resolution, **sethostent_r** allows a request for the use of a connected socket using TCP for queries. If the *StayOpen* parameter is non-zero, this sets the option to send all queries to the name server using TCP and to retain the connection after each call to **gethostbyname_r** or **gethostbyaddr_r**.

When using the **sethostent_r** subroutine to search the **/etc/hosts** file, **sethostent_r** opens and rewinds the **/etc/hosts** file. If the *StayOpen* parameter is non-zero, the hosts database is not closed after each call to **gethostbyname_r** or **gethostbyaddr_r**. It internally runs the **sethostent** command.

Parameters

Item	Description
StayOpenflag	When used in NIS name resolution and to search the local **/etc/hosts** file, it contains a value used to indicate whether to close the host file after each call to the **gethostbyname** and **gethostbyaddr** subroutines. A non-zero value indicates not to close the host file after each call, and a zero value allows the file to be closed. When used in DNS/BIND name resolution, a non-zero value retains the TCP connection after each call to **gethostbyname** and **gethostbyaddr**. A value of zero allows the connection to be closed.
ht_data	Points to the **hostent_data** structure.

Files

Item	Description
/etc/hosts	Contains the host name database.
/etc/netsvc.conf	Contains the name services ordering.
/etc/include/netdb.h	Contains the network database structure.

sethostid Subroutine

Purpose

Sets the unique identifier of the current host.

Library

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

Syntax

```
int sethostid ( HostID)
int HostID;
```

Description

The **sethostid** subroutine allows a calling process with a root user ID to set a new 32-bit identifier for the current host. The **sethostid** subroutine enables an application program to reset the host ID.

All applications containing the **sethostid** subroutine must be compiled with the **_BSD** macro 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
HostID	Specifies the unique 32-bit identifier for the current host.

Return Values

Upon successful completion, the **sethostid** subroutine returns a value of 0.

If the **sethostid** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *Communications Programming Concepts*.

Error Codes

The **sethostid** subroutine is unsuccessful if the following is true:

Error	Description
EPERM	The calling process did not have an effective user ID of root user.

Related information
Sockets Overview

sethostname Subroutine

Purpose

Sets the name of the current host.

Library

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

Syntax

```
int sethostname ( Name,  NameLength)
char *Name;
int NameLength;
```

Description

The **sethostname** subroutine sets the name of a host machine. Only programs with a root user ID can use this subroutine.

The **sethostname** subroutine allows a calling process with root user authority to set the internal host name of a machine on a network.

All applications containing the **sethostname** subroutine must be compiled with the **_BSD** macro 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
Name	Specifies the name of the host machine.
NameLength	Specifies the length of the *Name* array.

Return Values

Upon successful completion, the system returns a value of 0.

If the **sethostname** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *General Programming Concepts: Writing and Debugging Programs*.

Error Codes

The **sethostname** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EFAULT	The *Name* parameter or *NameLength* parameter gives an address that is not valid.
EPERM	The calling process did not have an effective root user ID.

Related reference
gethostname Subroutine
gethostid Subroutine
Related information
Sockets Overview
Understanding Network Address Translation

setnetent Subroutine

Purpose

Opens the **/etc/networks** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>
void setnetent (StayOpen)
int StayOpen;
```

Description

The **setnetent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **setnetent** subroutine opens the **/etc/networks** file and sets the file marker at the beginning of the file.

All applications containing the **setnetent** subroutine must be compiled with the **_BSD** macro 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
StayOpen	Contains a value used to indicate when to close the **/etc/networks** file. Specifying a value of 0 closes the **/etc/networks** file after each call to the **getnetent** subroutine. Specifying a nonzero value leaves the **/etc/networks** file open after each call.

Return Values

If an error occurs or the end of the file is reached, the **setnetent** subroutine returns a null pointer.

Files

Item	Description
/etc/networks	Contains official network names.

Related reference
endnetent Subroutine
getnetent Subroutine
getnetbyaddr Subroutine
Related information
Sockets Overview

setnetent_r Subroutine

Purpose

Opens the **/etc/networks** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>
int setnetent_r(StayOpenflag, net_data)
struct netent_data *net_data;
int StayOpenflag;
```

Description

The **setnetent_r** subroutine opens the **/etc/networks** file and sets the file marker at the beginning of the file.

Parameters

Item	Description
StayOpenflag	Contains a value used to indicate when to close the **/etc/networks** file. Specifying a value of 0 closes the **/etc/networks** file after each call to the **getnetent** subroutine. Specifying a nonzero value leaves the **/etc/networks** file open after each call.
net_data	Points to the **netent_data** structure.

Files

Item	Description
/etc/networks	Contains official network names.

setnetgrent_r Subroutine

Purpose

Handles the group network entries.

Library

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

Syntax

```
#include <netdb.h>
int setnetgrent_r(NetGroup,ptr)
char *NetGroup;
void **ptr;
```

Description

The **setnetgrent_r** subroutine functions the same as the **setnetgrent** subroutine.

The **setnetgrent_r** subroutine establishes the network group from which the **getnetgrent_r** subroutine will obtain members. This subroutine also restarts calls to the **getnetgrent_r** subroutine from the beginnning of the list. If the previous **setnetgrent_r** call was to a different network group, an **endnetgrent_r** call is implied. The **endnetgrent_r** subroutine frees the space allocated during the **getnetgrent_r** calls.

Parameters

Item	Description
NetGroup	Points to a network group.
ptr	Keeps the function threadsafe.

Return Values

The **setnetgrent_r** subroutine returns a 0 if successful and a -1 if unsuccessful.

Files

Item	Description
/etc/netgroup	Contains network groups recognized by the system.
/usr/include/netdb.h	Contains the network database structures.

setprotoent Subroutine

Purpose

Opens the **/etc/protocols** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>

void setprotoent (StayOpen)
int StayOpen;
```

Description

The **setprotoent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **setprotoent** subroutine opens the **/etc/protocols** file and sets the file marker to the beginning of the file.

All applications containing the **setprotoent** subroutine must be compiled with the **_BSD** macro 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
StayOpen	Indicates when to close the **/etc/protocols** file. Specifying a value of 0 closes the file after each call to **getprotoent**. Specifying a nonzero value allows the **/etc/protocols** file to remain open after each subroutine.

Return Values

The return value points to static data that is overwritten by subsequent calls.

Files

Item	Description
/etc/protocols	Contains the protocol names.

Related reference
endprotoent Subroutine

getprotobyname Subroutine

Related information

Sockets Overview

setprotoent_r Subroutine

Purpose

Opens the **/etc/protocols** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>

int setprotoent_r(StayOpenflag, proto_data);
int StayOpenflag;
struct protoent_data *proto_data;
```

Description

The **setprotoent_r** subroutine opens the **/etc/protocols** file and sets the file marker to the beginning of the file.

Parameters

Item	Description
StayOpenflag	Indicates when to close the **/etc/protocols** file. Specifying a value of 0 closes the file after each call to **getprotoent**. Specifying a nonzero value allows the **/etc/protocols** file to remain open after each subroutine.

Files

Item	Description
/etc/protocols	Contains the protocol names.

setservent Subroutine

Purpose

Opens **/etc/services** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>

void setservent ( StayOpen)
int StayOpen;
```

Description

The **setservent** subroutine is threadsafe. However, the return value points to static data that is overwritten by subsequent calls. This data must be copied to be saved for use by subsequent calls.

The **setservent** subroutine opens the **/etc/services** file and sets the file marker at the beginning of the file.

All applications containing the **setservent** subroutine must be compiled with the **_BSD** macro 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
StayOpen	Indicates when to close the **/etc/services** file. Specifying a value of 0 closes the file after each call to the **getservent** subroutine. Specifying a nonzero value allows the file to remain open after each call.

Return Values

If an error occurs or the end of the file is reached, the **setservent** subroutine returns a null pointer.

Files

Item	Description
/etc/services	Contains service names.

Related reference
endprotoent Subroutine
getprotobyname Subroutine
Related information
Sockets Overview

setservent_r Subroutine

Purpose

Opens **/etc/services** file and sets the file marker.

Library

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

Syntax

```
#include <netdb.h>

int setservent_r(StayOpenflag, serv_data)
int StayOpenflag;
struct servent_data serv_data;
```

Description

The **setservent_r** subroutine opens the **/etc/services** file and sets the file marker at the beginning of the file.

Parameters

Item	Description
StayOpenflag	Indicates when to close the **/etc/services** file. Specifying a value of 0 closes the file after each call to the **getservent** subroutine. Specifying a nonzero value allows the file to remain open after each call.
serv_data	Points to the **servent_data** structure.

Files

Item	Description
/etc/services	Contains service names.

setsockopt Subroutine

Purpose

Sets socket options.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
#include <sys/atmsock.h> /*Needed for SOCK_CONN_DGRAM socket type
only*/
```

```
int setsockopt
(Socket, Level, OptionName, OptionValue, OptionLength)
int  Socket,  Level,  OptionName;
const void * OptionValue;
socklen_t  OptionLength;
```

Description

The **setsockopt** subroutine sets options associated with a socket. Options can exist at multiple protocol levels. The options are always present at the uppermost socket level.

The **setsockopt** subroutine provides an application program with the means to control a socket communication. An application program can use the **setsockopt** subroutine to enable debugging at the protocol level, allocate buffer space, control time outs, or permit socket data broadcasts. The **/usr/include/sys/socket.h** file defines all the options available to the **setsockopt** subroutine.

When setting socket options, specify the protocol level at which the option resides and the name of the option.

Use the parameters *OptionValue* and *OptionLength* to access option values for the **setsockopt** subroutine. These parameters identify a buffer in which the value for the requested option or options is returned.

All applications containing the **setsockopt** subroutine must be compiled with the **_BSD** macro 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
Socket	Specifies the unique socket name.
Level	Specifies the protocol level at which the option resides. To set options at: **Socket level** Specifies the *Level* parameter as **SOL_SOCKET**. **Other levels** Supplies the appropriate protocol number for the protocol controlling the option. For example, to indicate that an option will be interpreted by the TCP protocol, set the *Level* parameter to the protocol number of TCP, as defined in the **netinet/in.h** file. Similarly, to indicate that an option will be interpreted by ATM protocol, set the *Level* parameter to NDDPROTO_ATM, as defined in **sys/atmsock.h**.
OptionName	Specifies the option to set. The *OptionName* parameter and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The **sys/socket.h** file defines the socket protocol level options. The **netinet/tcp.h** file defines the TCP protocol level options. The socket level options can be enabled or disabled; they operate in a toggle fashion. The following list defines socket protocol level options found in the **sys/socket.h** file: **SO_DEBUG** Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules. Set this option in one of the following ways at the command level: • Use the `sodebug` command, which turns on or off this option for existing sockets. • Specify `|DEBUG[=level]` in the `wait/nowait` field of a service in `inetd.conf` in order to turn on this option for the specific service. • Set the `sodebug_env` parameter to no, and specify `SODEBUG=level` in the process environment. This turns on or off this option for all subsequent sockets created by the process. The value for *level* can be either `min`, `normal`, or `detail`. **SO_REUSEADDR** Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port. **SO_REUSEADDR** allows an application to explicitly deny subsequent **bind** subroutine to the port/address of the socket with **SO_REUSEADDR** set. This allows an application to block other applications from binding with the **bind** subroutine. **SO_REUSEPORT** Specifies that the rules used in validating addresses supplied by a **bind** subroutine should allow reuse of a local port/address combination. Each binding of the port/address combination must specify the **SO_REUSEPORT** socket option **SO_CKSUMREV** Enables performance enhancements in the protocol layers. If the protocol supports this option, enabling causes the protocol to defer checksum verification until the user's data is moved into the user's buffer (on **recv**, **recvfrom**, **read**, or **recvmsg** thread). This can cause applications to be awakened when no data is available, in the case of a checksum error. In this case, EAGAIN is returned. Applications that set this option must handle the EAGAIN error code rturned from a receive call. **SO_KEEPALIVE** Monitors the activity of a connection by enabling or disabling the periodic transmission of ACK messages on a connected socket. The idle interval time can be designated using the TCP/IP **no** command. Broken connections are discussed in "Understanding Socket Types and Protocols" in *Communications Programming Concepts*.

Item	Description
OptionName	**SO_DONTROUTE** Does not apply routing on outgoing messages. Indicates that outgoing messages should bypass the standard routing facilities. Instead, they are directed to the appropriate network interface according to the network portion of the destination address. **SO_BROADCAST** Permits sending of broadcast messages. **SO_LINGER** Lingers on a **close** subroutine if data is present. This option controls the action taken when an unsent messages queue exists for a socket, and a process performs a **close** subroutine on the socket. If **SO_LINGER** is set, the system blocks the process during the **close** subroutine until it can transmit the data or until the time expires. If **SO_LINGER** is not specified and a **close** subroutine is issued, the system handles the call in a way that allows the process to continue as quickly as possible. The **sys/socket.h** file defines the **linger** structure that contains the **l_linger** member for specifying linger time interval. If linger time is set to anything but 0, the system tries to send any messages queued on the socket. The maximum value that the **l_linger** member can be set to is 65535. If the application has requested SPEC1170 compliant behavior by exporting the XPG_SUS_ENV environment variable , the linger time is *n* seconds; otherwise, the linger time is *n*/100 seconds (ticks), where *n* is the value of the **l_linger** member. **SO_OOBINLINE** Leaves received out-of-band data (data marked urgent) in line. **SO_SNDBUF** Sets send buffer size. **SO_RCVBUF** Sets receive buffer size. **SO_SNDLOWAT** Sets send low-water mark. **SO_RCVLOWAT** Sets receive low-water mark. **SO_SNDTIMEO** Sets send time out. This option is setable, but currently not used. **SO_RCVTIMEO** Sets receive time out. This option is setable, but currently not used. **SO_ERROR** Sets the retrieval of error status and clear. **SO_TYPE** Sets the retrieval of a socket type.

Item	Description
OptionName	The following list defines TCP protocol level options found in the **netinet/tcp.h** file: **TCP_CWND_IF** Increases the factor of the TCP congestion window (cwnd) during the congestion avoidance. The value must be in the range 0 - 100 (0 is disable). The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_CWND_DF** Decrease the factor of the TCP cwnd during the congestion avoidance. The value must be in the range 0 - 100 (0 is disable). The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_NOTENTER_SSTART** Avoids reentering the slow start after the retransmit timeout, which might reset the cwnd to the initial window size, instead of the size of the current slow-start threshold (ss_threshold) value or half of the maximum cwnd (max cwnd/2). The values are 1 for enable and 0 for disable. The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_ NOREDUCE_CWND_IN_FRXMT** Not decrease the cwnd size when in the fast retransmit phrase. The values are 1 for enable and 0 for disable. The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_ NOREDUCE_CWND_EXIT_FRXMT** Not decrease the cwnd size when exits the fast retransmit phrase. The values are 1 for enable and 0 for disable. The **tcp_cwnd_modified** network tunable option must be enabled. **TCP_KEEPCNT** Specifies the maximum number of keepalive packets to be sent to validate a connection. This socket option value is inherited from the parent socket. The default is 8. **TCP_KEEPIDLE** Specifies the number of seconds of idle time on a connection after which TCP sends a keepalive packet. This socket option value is inherited from the parent socket from the accept system call. The default value is 7200 seconds (14400 half seconds). **TCP_KEEPINTVL** Specifies the interval of time between keepalive packets. It is measured in seconds. This socket option value is inherited from the parent socket from the accept system call. The default value is 75 seconds (150 half seconds). **TCP_NODELAY** Specifies whether TCP should follow the Nagle algorithm for deciding when to send data. By default, TCP will follow the Nagle algorithm. To disable this behavior, applications can enable **TCP_NODELAY** to force TCP to always send data immediately. For example, **TCP_NODELAY** should be used when there is an application using TCP for a request/response.

Item	Description

OptionName	**TCP_RFC1323**
	Enables or disables RFC 1323 enhancements on the specified TCP socket. An application might contain the following lines to enable RFC 1323:
	`int on=1;`
	`setsockopt(s,IPPROTO_TCP,TCP_RFC1323,&on,`
	`sizeof(on));`
	TCP_STDURG
	Enables or disables RFC 1122 compliant urgent point handling. By default, TCP implements urgent pointer behavior compliant with the 4.2 BSD operating system, i.e., this option defaults to 0.
	TCP_NODELAYACK
	Specifies if TCP needs to send immediate acknowledgement packets to the sender. If this option is not set, TCP delays sending the acknowledgement packets by up to 200 ms. This allows the acknowledgements to be sent along with the data on a response and minimizes system overhead. Setting this TCP option might cause a slight increase in system overhead, but can result in higher performance for network transfers if the sender is waiting on the receiver's acknowledgements.
	TCP protocol level socket options are inherited from listening sockets to new sockets.
	The following list defines ATM protocol level options found in the **sys/atmsock.h** file:
	SO_ATM_PARAM
	Sets all ATM parameters. This socket option can be used instead of using individual sockets options described below. It uses the **connect_ie** structure defined in **sys/call_ie.h** file.
	SO_ATM_AAL_PARM
	Sets ATM AAL(Adaptation Layer) parameters. It uses the **aal_parm** structure defined in **sys/call_ie.h** file.
	SO_ATM_TRAFFIC_DES
	Sets ATM Traffic Descriptor values. It uses the **traffic** structure defined in **sys/call_ie.h** file.
	SO_ATM_BEARER
	Sets ATM Bearer capability. It uses the **bearer** structure defined in **sys/call_ie.h** file.
	SO_ATM_BHLI
	Sets ATM Broadband High Layer Information. It uses the **bhli** structure defined in **sys/call_ie.h** file.
	SO_ATM_BLLI
	Sets ATM Broadband Low Layer Information. It uses the **blli** structure defined in **sys/call_ie.h** file.
	SO_ATM_QOS
	Sets ATM Quality Of Service values. It uses the **qos_parm** structure defined in **sys/call_ie.h** file.
	SO_ATM_TRANSIT_SEL
	Sets ATM Transit Selector Carrier. It uses the **transit_sel** structure defined in **sys/call_ie.h** file.

OptionName	**SO_ATM_ACCEPT** Indicates acceptance of an incoming ATM call, which was indicated to the application via *ACCEPT* system call. This must be issues for the incoming connection to be fully established. This allows negotiation of ATM parameters. **SO_ATM_MAX_PEND** Sets the number of outstanding transmit buffers that are permitted before an error indication is returned to applications as a result of a transmit operation. This option is only valid for non best effort types of virtual circuits. OptionValue/OptionLength point to a byte which contains the value that this parameter will be set to. The following list defines **IPPROTO_TCP** protocol level options found in the **netinet/sctp.h** file: **SCTP_PEER_ADDR_PARAMS** Enables or disables heartbeats for an association and modifies the heartbeat interval of the association. This option uses the *sctp_paddrparams* structure defined in the **netinet/sctp.h** file. For *spp_address* field, AIX only supports wildcard address now. The **SPP_HB_ENABLE**, **SPP_HB_DISABLE**, and **SPP_HB_TIME_ISZERO** flags are supported for the *spp_flags* field. The *spp_hbinterval* field can be set to a minimum value of 50 milliseconds. **SCTP_MAXSEG** Sets the maximum size of any outgoing **SCTP DATA** chunk. If the message is larger than the specified size, the message is fragmented by SCTP into the specified size. It uses the *sctp_assoc_value* structure that is defined in the **netinet/sctp.h** file.

Item	Description
OptionValue	The *OptionValue* parameter takes an *Int* parameter. To enable a Boolean option, set the *OptionValue* parameter to a nonzero value. To disable an option, set the *OptionValue* parameter to 0. The following options enable and disable in the same manner: • **SO_DEBUG** • **SO_REUSEADDR** • **SO_KEEPALIVE** • **SO_DONTROUTE** • **SO_BROADCAST** • **SO_OOBINLINE** • **SO_LINGER** • **TCP_RFC1323**
OptionLength	The *OptionLength* parameter contains the size of the buffer pointed to by the *OptionValue* parameter.

Options at other protocol levels vary in format and name.

Item	Description
IP_DONTFRAG	Sets DF bit from now on for every packet in the IP header. To detect decreases in Path MTU, UDP applications use the **IP_DONTFRAG** option.
IP_FINDPMTU	Sets enable/disable PMTU discovery for this path. Protocol level path MTU discovery should be enabled for the discovery to happen.
IP_PMTUAGE	Sets the age of PMTU. Specifies the frequency of PMT reductions discovery for the session. Setting it to 0 (zero) implies infinite age and PMTU reduction discovery will not be attempted. This will replace the previously set PMTU age. The new PMTU age is effective after the currently set timer expires. Currently, this option is unused because UDP applications must set the **IP_DONTFRAG** socket to detect decreases in PMTU immediately.
IP_TTL	Sets the time-to-live field in the IP header for every packet. However, for raw sockets, the default MAXTTL value will be used while sending the messages irrespective of the value set using the **setsockopt** subroutine.
IP_HDRINCL	This option allows users to build their own IP header. It indicates that the complete IP header is included with the data and can be used only for raw sockets.
IP_ADD_MEMBERSHIP	Joins a multicast group as specified in the *OptionValue* parameter of the **ip_mreq** structure type.
IP_DROP_MEMBERSHIP	Leaves a multicast group as specified in the *OptionValue* parameter of the **ip_mreq** structure type.
IP_MULTICAST_IF	Permits sending of multicast messages on an interface as specified in the *OptionValue* parameter of the **ip_addr** structure type. An address of **INADDR_ANY** (0x000000000) removes the previous selection of an interface in the multicast options. If no interface is specified, the interface leading to the default route is used.
IP_MULTICAST_LOOP	Sets multicast loopback, determining whether or not transmitted messages are delivered to the sending host. An *OptionValue* parameter of the **char** type controls the loopback to be on or off.
IP_MULTICAST_TTL	Sets the time-to-live (TTL) for multicast packets. An *OptionValue* parameter of the **char** type sets the value of TTL ranging from 0 through 255.
IP_BLOCK_SOURCE	Blocks data from a given source to a given group.
IP_UNBLOCK_SOURCE	Unblocks a blocked source (to undo the **IP_BLOCK_SOURCE** operation).
IP_ADD_SOURCE_MEMBERSHIP	Joins a source-specific multicast group. If the host is a member of the group, accept data from the source; otherwise, join the group and accept data from the given source.

Item	Description
IP_DROP_SOURCE_MEMBERSHIP	Leaves a source-specific multicast group. Drops the source from the given multicast group list. To drop all sources of a given group, use the **IP_DROP_MEMBERSHIP** socket option.

Item	Description	Value
IPPROTO_IPV6	Restricts AF_INET6 sockets to IPv6 communications only.	**Option Type:** int (Boolean interpretation)
	Allows the user to set the outgoing hop limit for unicast IPv6 packets.	**Option Type:** int (x) **Option Value:** x < -1 Error EINVAL x == -1 Use kernel default 0 <= x <= 255 Use x x >= 256 Error EINVAL
	Allows the user to set the outgoing hop limit for multicast IPv6 packets.	**Option Type:** int (x) **Option Value:** Interpretation is same as IPV6_UNICAST_HOPS (listed above).
	Allows the user to specify the interface being used for outgoing multicast packets. If specified as 0, the system selects the outgoing interface.	**Option Type:** unsigned int (index of interface to use)
	If a multicast datagram is sent to a group that the sending host belongs to, a copy of the datagram is looped back by the IP layer for local delivery (if the option is set to 1). If the option is set to 0, a copy is not looped back.	**Option Type:** unsigned int
	Joins a multicast group on a specified local interface. If the interface index is specified as 0, the kernel chooses the local interface.	**Option Type:** **struct ipv6_mreq** as defined in the **netinet/in.h** file
	Leaves a multicast group on a specified interface.	**Option Type:** **struct ipv6_mreq** as defined in the **netinet/in.h** file
	Specifies that the kernel computes checksums over the data and the pseudo-IPv6 header for a raw socket. The kernel will compute the checksums for outgoing packets as well as verify checksums for incoming packets on that socket. Incoming packets with incorrect checksums will be discarded. This option is disabled by default.	**Option Type:** int **Option Value:** Offsets into the user data where the checksum result must be stored. This must be a positive even value. Setting the value to -1 will disable the option.
	Causes the destination IPv6 address and arriving interface index of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	**Option Type:** **int** (Boolean interpretation)
	Causes the hop limit of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	**Option Type:** **int** (Boolean interpretation)
	Causes the traffic class of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	**Option Type:** **int** (Boolean interpretation)
	Causes the routing header (if any) of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	**Option Type:** **int** (Boolean interpretation)
	Causes the hop-by-hop options header (if any) of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	**Option Type:** **int** (Boolean interpretation)
	Causes the destination options header (if any) of incoming IPv6 packets to be received as ancillary data on UDP and raw sockets.	**Option Type:** **int** (Boolean interpretation)
	Sets the source IPv6 address and outgoing interface index for all IPv6 packets being sent on this socket. This option can be cleared by doing a regular **setsockopt** with **ipi6_addr** being **in6addr_any** and **ipi6_ifindex** being 0.	**Option Type:** **struct in6_pktinfo** defined in the **netinet/in.h** file.
	Sets the next hop for outgoing IPv6 datagrams on this socket. This option can be cleared by doing a regular **setsockopt** with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.	**Option Type:** **struct sockaddr_in6** defined in the **netinet/in.h** file.

Item	Description	Value
	Sets the traffic class for outgoing IPv6 datagrams on this socket. To clear this option, the application can specify -1 as the value.	**Option Type:** int (x) **Option Value:** x < -1 Error EINVAL x == -1 Use kernel default 0 <= x <= 255 Use x x >= 256 Error EINVAL
	Sets the routing header to be used for outgoing IPv6 datagrams on this socket. This option can be cleared by doing a regular **setsockopt** with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.	**Option Type:** **struct ip6_rthdr** defined in the **netinet/ip6.h** file.
	Sets the hop-by-hop options header to be used for outgoing IPv6 datagrams on this socket. This option can be cleared by doing a regular **setsockopt** with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.	**Option Type:** **struct ip6_hbh** defined in the **netinet/ip6.h** file.
	Sets the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will follow a routing header (if present) and will also be used when there is no routing header specified. This option can be cleared by doing a regular **setsockopt** with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.	**Option Type:** **struct ip6_dest** defined in the **netinet/ip6.h** file.
	Sets the destination options header to be used for outgoing IPv6 datagrams on this socket. This header will precede a routing header (if present). If no routing header is specified, this option will be silently ignored. This option can be cleared by doing a regular **setsockopt** with a 0 length. Note that a memory pointer must still be supplied for the option value in this case.	**Option Type:** **struct ip6_dest** defined in the **netinet/ip6.h** file.
	Sets this option to control IPv6 path MTU discovery.	**Option Type:** int **Option Type:** -1 Performs path MTU discovery for unicast destinations, but does not perform it for multicast destinations.0 Always performs path MTU discovery.1 Always disables path MTU discovery and sends packets at the minimum MTU.
	Setting this option prevents fragmentation of outgoing IPv6 packets on this socket. If a packet is being sent that is larger than the outgoing interface MTU, the packet will be discarded.	**Option Type:** **int** (Boolean interpretation)
	Enables the receipt of IPV6_PATHMTU ancillary data items by setting this option.	**Option Type:** **int** (Boolean interpretation)
	Sets the address selection preferences for this socket.	**Option Type:** **int** **Option Value:** Combination of the IPV6_PREFER_SRC_* flags defined in **netinet/in.h**
	Joins the multicast group as specified in the *OptionValue* parameter of the **group_req** structure. If the specified interface index is 0, the kernel chooses the default interface.	**Option Type:** **struct group_req** as defined in the **netinet/in.h** file
	Leaves the multicast group as specified in the *OptionValue* parameter of the **group_req** structure.	**Option Type:** **struct group_req** as defined in the **netinet/in.h** file
	Blocks data from the specified source to the specified multicast group.	**Option Type:** **struct group_source_req** as defined in the **netinet/in.h** file
	Unblocks data from the specified source to the specified multicast group. The option is used to undo the **MCAST_BLOCK_SOURCE** operation.	**Option Type:** **struct group_source_req** as defined in the **netinet/in.h** file
	Joins a source-specific multicast group. If the host is already a member of the group, accept data from the specified source; otherwise, join the group and accept data from the specified source.	**Option Type:** **struct group_source_req** as defined in the **netinet/in.h** file

Item	Description	Value
	Leaves a source-specific multicast group. Leaves the specified source from the specified multicast group. To leave all sources of the multicast group, use the **IPV6_LEAVE_GROUP** or **MCAST_LEAVE_GROUP** socket option.	**Option Type:** **struct group_source_req** as defined in the **netinet/in.h** file

Item	Description	Value
IPPROTO_ICMPV6	Allows the user to filter ICMPV6 messages by the ICMPV6 type field. In order to clear an existing filter, issue a **setsockopt** call with zero length.	**Option Type:** The **icmp6_filter** structure defined in the **netinet/icmp6.h** file.

The following values (defined in the **/usr/include/netint/tcp.h** file) are used by the **setsockopt** subroutine to configure the **dacinet** functions.

Note: The DACinet facility is available only in a CAPP/EAL4+ configured AIX system.

```
tcp.h:#define TCP_ACLFLUSH    0x21    /* clear all DACinet ACLs */
tcp.h:#define TCP_ACLCLEAR    0x22    /* clear DACinet ACL */
tcp.h:#define TCP_ACLADD      0x23    /* Add to DACinet ACL */
tcp.h:#define TCP_ACLDEL      0x24    /* Delete from DACinet ACL */
tcp.h:#define TCP_ACLLS       0x25    /* List DACinet ACL */
tcp.h:#define TCP_ACLBIND     0x26    /* Set port number for TCP_ACLLS */
tcp.h:#define TCP_ACLGID      0x01    /* ID being added to ACL is a GID */
tcp.h:#define TCP_ACLUID      0x02    /* ID being added to ACL is a GID */
tcp.h:#define TCP_ACLSUBNET   0x04    /* address being added to ACL is a subnet */
tcp.h:#define TCP_ACLDENY     0x08    /* this ACL entry is for denying access */
```

Return Values

Upon successful completion, a value of 0 is returned.

If the **setsockopt** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

Item	Description
EBADF	The *Socket* parameter is not valid.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINVAL	The *OptionValue* parameter or the *OptionLength* parameter is invalid or the socket has been shutdown.
ENOBUFS	There is insufficient memory for an internal data structure.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
ENOPROTOOPT	The option is unknown.
EOPNOTSUPP	The option is not supported by the socket family or socket type.
EPERM	The user application does not have the permission to get or to set this socket option. Check the network tunable option

Examples

- To mark a socket for broadcasting:

```
int on=1;
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof(on));
```

- To turn on the **TCP_NODELAYACK** option, run the following:

```
int on=1;
setsockopt(s, IPPROTO_TCP, TCP_NODELAYACK, &on, sizeof(on));
```

Related reference
sendto Subroutine
bind Subroutine
Related information
no subroutine

setsourcefilter, getsourcefilter, setipv4sourcefilter, getipv4sourcefilter Subroutine

Purpose

Manage IP multicast source filters.

Library

Library (**libc.a**)

Syntax

```
#include <netinet/in.h>
int setsourcefilter(int socket, uint32_t interface,
                        struct sockaddr *group, socklen_t grouplen,
                        uint32_t fmode, uint_t numsrc,
                        struct sockaddr_storage *slist);

   int getsourcefilter(int socket, uint32_t interface,
                        struct sockaddr *group, socklen_t grouplen,
                        uint32_t *fmode, uint_t *numsrc,
                        struct sockaddr_storage *slist);

    int setipv4sourcefilter(int socket, struct in_addr interface,
                            struct in_addr group, uint32_t fmode,
                            uint32_t numsrc, struct in_addr *slist);

    int getipv4sourcefilter(int socket, struct in_addr interface,
                            struct in_addr group, uint32_t *fmode,
                            uint32_t *numsrc, struct in_addr *slist);
```

Description

The **setsourcefilter** and **setipv4sourcefilter** subroutines allow a socket to join a multicast group on an interface while excluding (fmode = MCAST_EXCLUDE) messages or accepting (fmode = MCAST_INCLUDE) messages from a number of senders listed in the slist table. The number of elements in the slist is specified by numsrc.

The **getsourcefilter** and **getipv4sourcefilter** subroutines provide information on existing source filter for a socket on a given interface and for a given multicast group. fmode, numsrc and slist are pointers to parameters which will contain the information returned by the subroutine. fmode will point to the type of filter returned: MCAST_EXCLUDE or MCAST_INCLUDE. On input, numsrc points to the maximum number of senders that the application is expecting. If there are more sources than requested, the subroutine returns only the first numsrc sources in slist and numsrc is set to indicate the total number of sources. slist contains the table of excluded or included senders depending on the type of the filter. Memory pointed by fmode, numsrc and slist must be allocated by the application. In particular, slist must point to a memory zone able to contain numsrc elements.

The **setipv4sourcefilter** and **getipv4sourcefilter** can only be used for AF_INET sockets.

The **setsourcefilter** and **getsourcefilter** can be used for AF_INET and AF_INET6 sockets.

Parameters

For setsourcefilter and setipv4sourcefilter:

Item	Description
socket	Specifies the unique socket name
interface	Specifies the local interface. For **setipv4sourcefilter** and **getipv4sourcefilter** an address configured on the interface must be specified. For **setsourcefilter** and **getsourcefilter**, the interface must be specified by its interface index.
group	Specifies the multicast group
fmode	Specifies if the elements contained in the slist must be excluded (MCAST_EXCLUDE) or included (MCAST_INCLUDE)
numsrc	Specifies the number of elements in slist
slist	Specifies the list of elements to exclude or include.

For getsourcefilter and getipv4sourcefilter:

Item	Description
socket	Specifies the unique socket name
interface	Specifies the local interface. For **setipv4sourcefilter** and **getipv4sourcefilter** an address configured on the interface must be specified. For **setsourcefilter** and **getsourcefilter** the interface must be specified by its interface index.
group	Specifies the multicast group
fmode	Specifies a pointer to the type of element returned in slist. MCAST_EXCLUDE for a list of excluded elements MCAST_INCLUDE for a list of excluded elements.
numsrc	On input, specifies the number of elements that can be returned in slist. On output, contains the total number of sources for this filter
slist	Contains the list of elements returned.

Return Values

Upon successful completion, the subroutine returns 0.

If unsuccessful, the subroutine returns -1 and errno is set accordingly.

shutdown Subroutine

Purpose

Shuts down all socket send and receive operations.

Library

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

Syntax

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

```
int shutdown ( Socket,  How)
int Socket, How;
```

Description

The **shutdown** subroutine disables all receive and send operations on the specified socket.

All applications containing the **shutdown** subroutine must be compiled with the **_BSD** macro 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
Socket	Specifies the unique name of the socket.
How	Specifies the type of subroutine shutdown. Use the following values: **0** Disables further receive operations. **1** Disables further send operations. **2** Disables further send operations and receive operations.

Return Values

Upon successful completion, a value of 0 is returned.

If the **shutdown** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *General Programming Concepts: Writing and Debugging Programs*.

Error Codes

The **shutdown** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
EINVAL	The *How* parameter is invalid.
ENOTCONN	The socket is not connected.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.

Files

Item	Description
/usr/include/sys/socket.h	Contains socket definitions.
/usr/include/sys/types.h	Contains definitions of unsigned data types.

Related reference
getsockopt Subroutine
recv Subroutine

Related information
read subroutine
Sockets Overview

SLPAttrCallback Subroutine

Purpose

Returns the same callback type as the **SLPFindAttrs()** function.

Syntax

```
typedef SLPBoolean SLPAttrCallback(SLPHandle hSLP,
                                   const char* pcAttrList,
                                   SLPError errCode,
                                   void *pvCookie);
```

Description

The **SLPAttrCallback** type is the type of the callback function parameter to the **SLPFindAttrs()** function.

The *pcAttrList* parameter contains the requested attributes as a comma-separated list (or is empty if no attributes matched the original tag list).

Parameters

Item	Description
hSLP	The **SLPHandle** used to initiate the operation.
pcAttrList	A character buffer containing a comma-separated, null-terminated list of attribute ID/value assignments, in SLP wire format: "`(attr-id=attr-value-list)`"
errCode	An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than **SLP_OK**, then the API library can choose to terminate the outstanding operation.
pvCookie	Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

Return Values

The client code should return **SLP_TRUE** if more data is desired; otherwise **SLP_FALSE** is returned.

Related information
/etc/slp.conf File
Service Location Protocol (SLP) API

SLPClose Subroutine

Purpose

Frees all resources associated with the handle.

Syntax

```
void SLPClose(SLPHandle hSLP);
```

Description

The **SLPClose** subroutine frees all resources associated with the handle. If the handle was invalid, the function returns silently. Any outstanding synchronous or asynchronous operations are cancelled so that their callback functions will not be called any further.

Parameters

Item	Description
hSLP	The **SLPHandle** handle returned from a call to **SLPOpen()**.

Related information

/etc/slp.conf File

Service Location Protocol (SLP) API

SLPDereg Subroutine

Purpose

Deregisters the advertisement for URL in all scopes and locales.

Syntax

```
SLPError SLPReg(hSLP, pcURL, callback, pvCookie)
SLPHandle hSLP;
const char  *pcURL;
SLPRegReport callback;
void *pvCookie;
```

Description

The **SLPDereg** subroutine deregisters the advertisement for the URL specified by the *pcURL* parameter in all scopes where the service is registered and in all language locales. The deregistration is not confined to the **SLPHandle** locale. Deregistration takes place in all locales.

Parameters

Item	Description
hSLP	The language-specific **SLPHandle** handle used for deregistration of services.
pcURL	The URL that needs to be deregistered.
callback	A callback function through which the results of the operation are reported.
pvCookie	The memory passed to callback code from the client. The parameter can be set to NULL.

Return Values

Item	Description
SLP_OK	The subroutine has run successfully.

Item	Description
SLPError	An error occurred.

Related information
/etc/slp.conf subroutine
Service Location Protocol (SLP) APIs

SLPEscape Subroutine

Purpose

Processes an input string and escapes any characters reserved for SLP.

Syntax

```
SLPError SLPEscape(const char* pcInbuf,
                   char** ppcOutBuf,
                   SLPBoolean isTag);
```

Description

The **SLPEscape** subroutine processes the input string in *pcInbuf* and escapes any characters reserved for SLP. If the *isTag* parameter is **SLPTrue**, **SLPEscape** looks for bad tag characters and signals an error if any are found by returning the **SLP_PARSE_ERROR** code. The results are put into a buffer allocated by the API library and returned in the *ppcOutBuf* parameter. This buffer should be deallocated using **SLPFree()** when the memory is no longer needed.

Parameters

Item	Description
pcInbuf	Pointer to the input buffer to process for escape characters.
ppcOutBuf	Pointer to a pointer for the output buffer with the characters reserved for SLP escaped. Must be freed using **SLPFree()** when the memory is no longer needed.
isTag	When true, the input buffer is checked for bad tag characters.

Return Values

The **SLPEscape** subroutine returns **SLP_PARSE_ERROR** if any characters are bad tag characters and the *isTag* flag is true; otherwise, it returns **SLP_OK**, or the appropriate error code if another error occurs.

Related information
/etc/slp.conf File
Service Location Protocol (SLP) API

SLPFindAttrs Subroutine

Purpose

Returns service attributes that match the attribute IDs for the indicated service URL or service type.

Syntax

```
SLPError SLPFindAttrs(SLPHandle   hSLP,
                      const char *pcURLOrServiceType,
                      const char *pcScopeList,
                      const char *pcAttrIds,
                      SLPAttrCallback callback,
                      void *pvCookie);
```

Description

The **SLPFindAttrs** subroutine returns service attributes matching the attribute IDs for the indicated service URL or service type. If **pcURLOrServiceType** is a service URL, the attribute information returned is for that particular advertisement in the language locale of the *SLPHandle*.

If **pcURLOrServiceType** is a service type name (including naming authority if any), then the attributes for all advertisements of that service type are returned regardless of the language of registration. Results are returned through the *callback*.

The result is filtered with an SLP attribute request filter string parameter. If the filter string is the empty string (" "), all attributes are returned.

Parameters

Item	Description
hSLP	The **SLPHandle** on which to search for attributes.
pcURLOrServiceType	The service URL or service type. Cannot be the empty string.
pcScopeList	A pointer to a char containing a comma-separated list of scope names. Cannot be the empty string, " ".
pcAttrIds	The filter string indicating which attribute values to return. Use the empty string (" ") to indicate all values. Wildcards matching all attribute IDs having a particular prefix or suffix are also possible.
callback	A callback function through which the results of the operation are reported.
pvCookie	Memory passed to the callback code from the client. Can be NULL.

Return Values

If **SLPFindAttrs** is successful, it returns **SLP_OK**. If an error occurs in starting the operation, one of the **SLPError** codes is returned.

Related information

/etc/slp.conf File

Service Location Protocol (SLP) API

SLPFindScopes Subroutine

Purpose

Sets the *ppcScopeList* parameter to point to a comma-separated list that includes all available scope values.

Syntax

```
SLPError SLPFindScopes(SLPHandle hSLP,
                       char** ppcScopeList);
```

Description

The **SLPFindScopes** subroutine sets the *ppcScopeList* parameter to point to a comma-separated list that includes all available scope values. If there is any order to the scopes, preferred scopes are listed before less desirable scopes. There is always at least one name in the list, the default scope, **DEFAULT**.

Parameters

Item	Description
hSLP	The **SLPHandle** on which to search for scopes.
ppcScopeList	A pointer to a char pointer into which the buffer pointer is placed upon return. The buffer is null terminated. The memory should be freed by calling **SLPFree()**.

Return Values

If no error occurs, **SLPFindScopes** returns **SLP_OK**; otherwise, it returns the appropriate error code.

Related information

/etc/slp.conf File

Service Location Protocol (SLP) API

SLPFindSrvs Subroutine

Purpose

Issues the query for services on the language-specific **SLPHandle** and returns the results through the *callback*.

Syntax

```
SLPError SLPFindSrvs(SLPHandle  hSLP,
                        const char *pcServiceType,
                        const char *pcScopeList,
                        const char *pcSearchFilter,
                        SLPSrvURLCallback callback,
                        void *pvCookie);
```

Description

The **SLPFindSrvs** subroutine issues the query for services on the language-specific **SLPHandle** and returns the results through the callback. The parameters determine the results

Parameters

Item	Description
hSLP	The language-specific **SLPHandle** on which to search for services.
pcServiceType	The Service Type String, including authority string if any, for the request, which can be discovered using **SLPSrvTypes()**. This could be, for example, `"service:printer:lpr"` or `"service:nfs"`. This cannot be the empty string (`""`).
pcScopeList	A pointer to a char containing a comma-separated list of scope names. This cannot be the empty string (`""`).

Item	Description
pcSearchFilter	A query formulated of attribute pattern matching expressions in the form of a LDAPv3 Search Filter. If this filter is empty (""), all services of the requested type in the specified scopes are returned.
callback	A callback function through which the results of the operation are reported.
pvCookie	Memory passed to the callback code from the client. Can be NULL.

Return Values

If **SLPFindSrvs** is successful, it returns **SLP_OK**. If an error occurs in starting the operation, one of the **SLPError** codes is returned.

Related information

/etc/slp.conf File

Service Location Protocol (SLP) API

SLPFindSrvTypes Subroutine

Purpose

Issues an SLP service type request.

Syntax

```
SLPError SLPFindSrvTypes(SLPHandle    hSLP,
                         const char  *pcNamingAuthority,
                         const char  *pcScopeList,
                         SLPSrvTypeCallback callback,
                         void *pvCookie);
```

Description

The **SLPFindSrvType()** subroutine issues an SLP service type request for service types in the scopes indicated by the **pcScopeList**. The results are returned through the *callback* parameter. The service types are independent of language locale, but only for services registered in one of the scopes and for the naming authority indicated by *pcNamingAuthority*.

If the naming authority is "*", then results are returned for all naming authorities. If the naming authority is the empty string, "", then the default naming authority, "IANA", is used. "IANA" is not a valid naming authority name, and it returns a **PARAMETER_BAD** error when it is included explicitly.

The service type names are returned with the naming authority intact. If the naming authority is the default (that is, the empty string), then it is omitted, as is the separating ".". Service type names from URLs of the **service:** scheme are returned with the "service:" prefix intact.

Parameters

Item	Description
hSLP	The **SLPHandle** on which to search for types.
pcNamingAuthority	The naming authority to search. Use "*" for all naming authorities and the empty string, "", for the default naming authority.
pcScopeList	A pointer to a char containing a comma-separated list of scope names to search for service types. Cannot be the empty string, "".

Item	Description
callback	A callback function through which the results of the operation are reported.
pvCookie	Memory passed to the callback code from the client. Can be NULL.

Return Values

If **SLPFindSrvTypes** is successful, it returns **SLP_OK**. If an error occurs in starting the operation, one of the **SLPError** codes is returned.

Related information
/etc/slp.conf File
Service Location Protocol (SLP) API

SLPFree Subroutine

Purpose

Frees memory returned from **SLPParseSrvURL()**, **SLPFindScopes()**, **SLPEscape()**, and **SLPUnescape()**.

Syntax

```
void SLPFree(void* pvMem);
```

Description

The **SLPFree** subroutine frees memory returned from **SLPParseSrvURL()**, **SLPFindScopes()**, **SLPEscape()**, and **SLPUnescape()**.

Parameters

Item	Description
pvMem	A pointer to the storage allocated by the **SLPParseSrvURL()**, **SLPEscape()**, **SLPUnescape()**, or **SLPFindScopes()** function. Ignored if NULL.

Related information
/etc/slp.conf File
Service Location Protocol (SLP) API

SLPGetProperty Subroutine

Purpose

Returns the value of the corresponding SLP property name.

Syntax

```
const char* SLPGetProperty(const char* pcName);
```

Description

The **SLPGetProperty** subroutine returns the value of the corresponding SLP property name. The returned string is owned by the library and *must not* be freed.

Parameters

Item	Description
pcName	Null-terminated string with the property name.

Return Values

If no error, the **SLPGetProperty** subroutine returns a pointer to a character buffer containing the property value. If the property was not set, the subroutine returns the default value. If an error occurs, it returns NULL. The returned string *must not* be freed.

Related information

/etc/slp.conf File

Service Location Protocol (SLP) API

SLPOpen Subroutine

Purpose

Returns an **SLPHandle** handle that encapsulates the language locale for SLP requests.

Syntax

```
SLPError SLPOpen(const char *pcLang, SLPBoolean isAsync, SLPHandle
   *phSLP);
```

Description

The **SLPOpen** subroutine returns an **SLPHandle** handle in the *phSLP* parameter for the language locale passed in as the *pcLang* parameter. The client indicates if operations on the handle are to be synchronous or asynchronous through the *isAsync* parameter. The handle encapsulates the language locale for SLP requests issued through the handle, and any other resources required by the implementation. However, SLP properties are not encapsulated by the handle; they are global. The return value of the function is an **SLPError** code indicating the status of the operation. Upon failure, the *phSLP* parameter is NULL.

Implementation Specifics

An **SLPHandle** can only be used for one SLP API operation at a time. If the original operation was started asynchronously, any attempt to start an additional operation on the handle while the original operation is pending results in the return of an **SLP_HANDLE_IN_USE** error from the API function. The **SLPClose()** API function terminates any outstanding calls on the handle. If an implementation is unable to support an asynchronous (resp. synchronous) operation, because of memory constraints or lack of threading support, the **SLP_NOT_IMPLEMENTED** flag might be returned when the *isAsync* flag is **SLP_TRUE** (resp. **SLP_FALSE**).

Parameters

Item	Description
pcLang	A pointer to an array of characters (AIX supports "en" only).

Item	Description
isAsync	An **SLPBoolean** indicating whether the **SLPHandle** should be opened for asynchronous operation or not. AIX supports synchronous operation only.
phSLP	A pointer to an **SLPHandle**, in which the open **SLPHandle** is returned. If an error occurs, the value upon return is NULL.

Return Values

If **SLPOpen** is successful, it returns **SLP_OK** and an **SLPHandle** handle in the *phSLP* parameter for the language locale passed in as the *pcLang* parameter.

Error Codes

Item	Description
SLPError	Indicates the status of the operation

Related information
/etc/slp.conf File
Service Location Protocol (SLP) API

SLPParseSrvURL Subroutine

Purpose

Parses the URL passed in as the argument into a service URL structure and returns it in the *ppSrvURL* pointer.

Syntax

```
SLPError SLPParseSrvURL(char *pcSrvURL
                        SLPSrvURL** ppSrvURL);
```

Description

The **SLPParseSrvURL** subroutine parses the URL passed in as the argument into a service URL structure and returns it in the *ppSrvURL* pointer. If a parse error occurs, returns **SLP_PARSE_ERROR**. The input buffer *pcSrvURL* is destructively modified during the parse and used to fill in the fields of the return structure. The structure returned in *ppSrvURL* should be freed with **SLPFreeURL()**. If the URL has no service part, the **s_pcSrvPart** string is the empty string (""), not NULL. If *pcSrvURL* is not a `service:` URL, then the **s_pcSrvType** field in the returned data structure is the URL's scheme, which might not be the same as the service type under which the URL was registered. If the transport is IP, the **s_pcTransport** field is the empty string. If the transport is not IP or there is no port number, the **s_iPort** field is 0.

Parameters

Item	Description
pcSrvURL	A pointer to a character buffer containing the null-terminated URL string to parse. It is destructively modified to produce the output structure.
ppSrvURL	A pointer to a pointer for the **SLPSrvURL** structure to receive the parsed URL. The memory should be freed by a call to **SLPFree()** when no longer needed.

Return Values

If no error occurs, the return value is **SLP_OK**. Otherwise, the appropriate error code is returned.

Related information
/etc/slp.conf File
Service Location Protocol (SLP) API

SLPReg Subroutine

Purpose

Registers the services on the language-specific **SLPHandle** handle and returns the results through the callback.

Syntax

```
SLPError SLPReg (hSLP, pcSrvURL,
usLifetime, pcSrvType,
 pcAttrs, fresh,
callback, pvCookie)
SLPHandle hSLP;
const char  *pcSrvURL;
const unsigned short usLifetime;
const char  *pcSrvType;
const char  *pcAttrs;
SLPBoolean  fresh;
SLPRegReport callback;
void *pvCookie;
```

Description

The **SLPReg** subroutine registers the URL specified by the *pcSrvURL* parameter having the *usLifeTime* lifetime with the attribute list specified by the *pcAttrs* parameter. The attribute list is a comma-separated list of attributes. The *pcSrvType* parameter is the service type name and can be included in the **scheme** service URL that are not in the service. In the case of the **scheme** service URL with service, the *pcSrvType* parameter is ignored. The *fresh* flag specifies that this registration is a new or an update-only registration. If the *fresh* parameter is set to SLP_TRUE, the registration replaces existing registrations. If the *fresh* parameter is set to SLP_FALSE, the registration only updates existing registrations. The *usLifeTime* parameter must be nonzero and less than or equal to SLP_LIFETIME_MAXIMUM. The registration takes place in the language locale of **hhSLP** handle.

Parameters

Item	Description
hSLP	The language-specific **SLPHandle** handle on which to register the services.
pcSrvURL	The URL that needs to be registered.
usLifetime	The time after which the registered URL will expire.
pcSrvType	Specifies the service type name that can be included in the service URL, which is not in the **scheme** service.
pcAttrs	The comma-separated list of attributes to be registered along with the service URL.
fresh	If the *fresh* parameter is set to SLP_TRUE, the registration is new; if the *fresh* parameter is set to SLP_FALSE, this registration updates an existing registration.
callback	A callback function through which the results of the operation are reported.

Item	Description
pvCookie	The memory passed to callback code from the client. The parameter can be set to NULL.

Return Values

Item	Description
SLP_OK	The subroutine has run successfully.
SLPError	An error occurred.

Related information
/etc/slp.conf subroutine
Service Location Protocol (SLP) APIs

SLPRegReport Callback Subroutine

Introduction

Returns the same callback type as the **SLPReg** and **SLPDereg** subroutines.

Syntax

```
typedef void SLPRegReport (hSLP, errCode, pvCookie)
SLPHandle hSLP;
SLPError errCode;
void *pvCookie;
```

Description

The **SLPSrvURLCallback** type is the type of the callback subroutine parameter to the **SLPFindSrvs** subroutine.

Parameters

Item	Description
hSLP	The **SLPHandle** handle used to initiate the operation.
errCode	An error code indicating that an error occurred during the operation. The callback must check this error code before processing the parameters. If the error code is not SLP_OK, the API library can choose to terminate the outstanding operation.
pvCookie	The memory passed down from the client code that calls the original API function at the start of the operation. The parameter can be set to NULL.

Return Values

Item	Description
SLP_TRUE	More data is necessary.
SLP_FALSE	No additional data is necessary.

SLPSrvTypeCallback Subroutine

Purpose

Returns the same callback type as the **SLPFindSrvTypes()** function.

Syntax

```
typedef SLPBoolean SLPSrvTypeCallback(SLPHandle hSLP,
                                      const char* pcSrvTypes,
                                      SLPError errCode,
                                      void *pvCookie);
```

Description

The **SLPSrvTypeCallback** type is the type of the callback function parameter to the **SLPFindSrvTypes()** function.

Parameters

Item	Description
hSLP	The **SLPHandle** used to initiate the operation.
pcSrvTypes	A character buffer containing a comma-separated, null-terminated list of service types.
errCode	An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than **SLP_OK**, then the API library can choose to terminate the outstanding operation.
pvCookie	Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

Return Values

The client code should return **SLP_TRUE** if more data is desired; otherwise **SLP_FALSE** is returned.

SLPSrvURLCallback Subroutine

Purpose

Returns the same callback type as the **SLPFindSrvs()** function.

Syntax

```
typedef SLPBoolean SLPSrvURLCallback(SLPHandle hSLP,
                                     const char* pcSrvURL,
                                     unsigned short sLifetime,
                                     SLPError errCode,
                                     void *pvCookie);
```

Description

The **SLPSrvURLCallback** type is the type of the callback function parameter to the **SLPFindSrvs()** function.

Parameters

Item	Description
hSLP	The **SLPHandle** used to initiate the operation.
pcSrvURL	A character buffer containing the returned service URL.
sLifetime	An unsigned short giving the lifetime of the service advertisement, in seconds. The value must be an unsigned integer less than or equal to **SLP_LIFETIME_MAXIMUM**.
errCode	An error code indicating if an error occurred during the operation. The callback should check this error code before processing the parameters. If the error code is other than **SLP_OK**, then the API library can choose to terminate the outstanding operation.
pvCookie	Memory passed down from the client code that called the original API function, starting the operation. Can be NULL.

Return Values

The client code should return **SLP_TRUE** if more data is desired; otherwise **SLP_FALSE** is returned.

Related information
/etc/slp.conf File
Service Location Protocol (SLP) API

SLPUnescape Subroutine

Purpose

Processes an input string and unescapes any characters reserved for SLP.

Syntax

```
SLPError SLPUnescape(const char* pcInbuf,
                     char** ppcOutBuf,
                     SLPBoolean isTag);
```

Description

The **SLPUnescape** subroutine processes the input string in *pcInbuf* and unescapes any characters reserved for SLP. If the *isTag* parameter is **SLPTrue**, **SLPUnescape** looks for bad tag characters and signals an error if any are found by returning the **SLP_PARSE_ERROR** code. No transformation is performed if the input string is opaque. The results are put into a buffer allocated by the API library and returned in the *ppcOutBuf* parameter. This buffer should be deallocated using **SLPFree()** when the memory is no longer needed.

Parameters

Item	Description
pcInbuf	Pointer to the input buffer to process for escape characters.
ppcOutBuf	Pointer to a pointer for the output buffer with the characters reserved for SLP escaped. Must be freed using **SLPFree()** when the memory is no longer needed.
isTag	When true, the input buffer is checked for bad tag characters.

Return Values

The **SLPUnescape** subroutine returns **SLP_PARSE_ERROR** if any characters are bad tag characters and the *isTag* flag is true; otherwise, it returns **SLP_OK**, or the appropriate error code if another error occurs.

Related information

/etc/slp.conf File

Service Location Protocol (SLP) API

socket Subroutine

Purpose

Creates an end point for communication and returns a descriptor.

Library

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

Syntax

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socketvar.h>
int socket ( AddressFamily,  Type,  Protocol)
int AddressFamily, Type, Protocol;
```

Description

The **socket** subroutine creates a socket in the specified *AddressFamily* and of the specified type. A protocol can be specified or assigned by the system. If the protocol is left unspecified (a value of 0), the system selects an appropriate protocol from those protocols in the address family that can be used to support the requested socket type.

The **socket** subroutine returns a descriptor (an integer) that can be used in later subroutines that operate on sockets.

Socket level options control socket operations. The **getsockopt** and **setsockopt** subroutines are used to get and set these options, which are defined in the **/usr/include/sys/socket.h** file.

Parameters

Item	Description
AddressFamily	Specifies an address family with which addresses specified in later socket operations should be interpreted. The **/usr/include/sys/socket.h** file contains the definitions of the address families. Commonly used families are: **AF_UNIX** Denotes the operating system path names. **AF_INET** Denotes the ARPA Internet addresses. **AF_INET6** Denotes the **IPv6** and **IPv4** addresses. **AF_NS** Denotes the XEROX Network Systems protocol. **AF_BYPASS** Denotes the kernel-bypass protocol domain (for example, the protocols that operate on the InfiniBand domain).
Type	Specifies the semantics of communication. The **/usr/include/sys/socket.h** file defines the socket types. The operating system supports the following types: **SOCK_STREAM** Provides sequenced, two-way byte streams with a transmission mechanism for out-of-band data. **SOCK_DGRAM** Provides datagrams, which are connectionless messages of a fixed maximum length (usually short). **SOCK_RAW** Provides access to internal network protocols and interfaces. This type of socket is available only to the root user, or to non-root users who have the CAP_NUMA_ATTACH capability. (For non-root raw socket access, the CAP_NUMA_ATTACH capability, along with CAP_PROPAGATE, is assigned using the **chuser** command. For more information about the **chuser** command, see **chuser Command** in *Commands Reference, Volume 1*.) **SOCK_SEQPACKET** Provides sequenced, reliable, and unduplicated flow of information. This type of socket is used for UDP-style socket creation in case of Stream Control Transmission Protocol and Reliable Datagram Sockets (RDS) Protocol.
Protocol	Specifies a particular protocol to be used with the socket. Specifying the *Protocol* parameter of 0 causes the **socket** subroutine to default to the typical protocol for the requested type of returned socket. For SCTP sockets, the protocol parameter is IPPROTO_SCTP. For RDS sockets, the *Protocol* parameter is BYPASSPROTO_RDS.

Return Values

Upon successful completion, the **socket** subroutine returns an integer (the socket descriptor).

If the **socket** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable see Error Notification Object Class in *General Programming Concepts: Writing and Debugging Programs*.

Error Codes

The **socket** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EMFILE	The per-process descriptor table is full.
ENOBUFS	Insufficient resources were available in the system to complete the call.
ESOCKTNOSUPPORT	The socket in the specified address family is not supported.

Examples

The following program fragment illustrates the use of the **socket** subroutine to create a datagram socket for on-machine use:

```
s = socket(AF_UNIX, SOCK_DGRAM,0);
```

Implementation Specifics

The socket subroutine is part of Base Operating System (BOS) Runtime.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Related reference

accept Subroutine

bind Subroutine

getsockname Subroutine

Related information

ioctl subroutine

Initiating Internet Stream Connections Example Program

socketpair Subroutine

Purpose

Creates a pair of connected sockets.

Library

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

Syntax

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

```
int socketpair (Domain, Type, Protocol, SocketVector[0])
int  Domain,  Type,  Protocol;
int  SocketVector[2];
```

Description

The **socketpair** subroutine creates an unnamed pair of connected sockets in a specified domain, of a specified type, and using the optionally specified protocol. The two sockets are identical.

Note: Create sockets with this subroutine only in the **AF_UNIX** protocol family.

The descriptors used in referencing the new sockets are returned in the *SocketVector*[**0**] and *SocketVector*[**1**] parameters.

The **/usr/include/sys/socket.h** file contains the definitions for socket domains, types, and protocols.

All applications containing the **socketpair** subroutine must be compiled with the **_BSD** macro set to a value of 43 or 44. Socket applications must include the BSD **libbsd.a** library.

Parameters

Item	Description
Domain	Specifies the communications domain within which the sockets are created. This subroutine does not create sockets in the Internet domain.
Type	Specifies the communications method, whether **SOCK_DGRAM** or **SOCK_STREAM**, that the socket uses.
Protocol	Points to an optional identifier used to specify which standard set of rules (such as UDP/IP and TCP/IP) governs the transfer of data.
SocketVector	Points to a two-element vector that contains the integer descriptors of a pair of created sockets.

Return Values

Upon successful completion, the **socketpair** subroutine returns a value of 0.

If the **socketpair** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable.

Error Codes

If the **socketpair** subroutine is unsuccessful, it returns one of the following errors codes:

Error	Description
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EFAULT	The *SocketVector* parameter is not in a writable part of the user address space.
EMFILE	This process has too many descriptors in use.
ENFILE	The maximum number of files allowed are currently open.
ENOBUFS	Insufficient resources were available in the system to perform the operation.
EOPNOTSUPP	The specified protocol does not allow the creation of socket pairs.
EPROTONOSUPPORT	The specified protocol cannot be used on this system.
EPROTOTYPE	The socket type is not supported by the protocol.

Related information

Socketpair Communication Example Program,

Sockets Overview,

socks5_getserv Subroutine

Purpose

Return the address of the SOCKSv5 server (if any) to use when connecting to a given destination.

Library

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

Syntax

```
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
```

```
struct sockaddr * socks5_getserv (Dst, DstLen)
struct sockaddr *Dst;
size_t DstLen;
```

Description

The **socks5_getserv** subroutine determines which (if any) SOCKSv5 server should be used as an intermediary when connecting to the address specified in *Dst*.

The address returned in *Dst* may be IPv4 or IPv6 or some other family. The user should check the address family before using the returned data.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Dst	Specifies the external address of the target socket to use as a key for looking up the appropriate SOCKSv5 server.
DstLength	Specifies the length of the address structure in *Dst*.

Return Values

- Upon successful lookup, the **socks_getserv** subroutine returns a reference to a sockaddr struct.
- If the **socks5tcp_connect** subroutine is unsuccessful in finding a server, for any reason, a value of NULL is returned. If an error occurred, an error code, indicating the generic system error, is moved into the **errno** global variable.

Error Codes (placed in errno)

The **socks5_getserv** subroutine is unsuccessful if no server is indicated or if any of the following errors occurs:

Error	Description
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EFAULT	The *Dst* parameter is not in a writable part of the user address space.
EINVAL	One or more of the specified arguments is invalid.
ENOMEM	The *Dst* parameter is not large enough to hold the server address.

Examples

The following program fragment illustrates the use of the **socks5_getserv** subroutine by a client to request a connection from a server's socket.

```
struct sockaddr_in6 dst;

struct sockaddr *srv;
.
.
.
srv = socks5_getserv((struct sockaddr*)&dst, sizeof(dst));

if (srv !=NULL)  {

     /* Success: srv should be used as the socks5 server */

} else {
     /* Failure: no server could be returned.  check errno */

}
```

Related reference
connect Subroutine
Related information
Sockets Overview
SOCKS5C_CONFIG Environment Variable

/etc/socks5c.conf File

Purpose

Contains mappings between network destinations and SOCKSv5 servers.

Description

The **/etc/socks5c.conf** file contains basic mappings between network destinations (hosts or networks) and SOCKSv5 servers to use when accessing those destinations. This is an ASCII file that contains records for server mappings. Text following a pound character ('#') is ignored until the end of line. Each record appears on a single line and is the following format:

<destination>[*/<prefixlength>*] *<server>*[*:<port>*]

You must separate fields with whitespace. Records are separated by new-line characters. The fields and modifiers in a record have the following values:

Item	Description
destination	Specifies a network destination; *destination* may be either a name fragment or a numeric address (with optional *prefixlength*). If *destination* is an address, it may be either IPv4 or IPv6.
prefixlength	If specified, indicates the number of leftmost (network order) bits of an address to use when comparing to this record. Only valid if *destination* is an address. If not specified, all bits are used in comparisons.
server	Specifies the SOCKSv5 server associated with *destination*. If *server* is "NONE" (must be all uppercase), this record indicates that target addresses matching *destination* should not use any SOCKSv5 server, but rather be contacted directly.
port	If specified, indicates the port to use when contacting *server*. If not specified, the default of 1080 is assumed. **Note:** Server address in IPv6 format **must** be followed by a port number.

If a name fragment *destination* is present in **/etc/socks5c.conf**, all target addresses is SOCKSv5 operations will be converted into hostnames for name comparison (in addition to numeric comparisons with numeric records). The resulting hostname is considered to match if the last characters in the hostname match the specified name fragment.

When using this configuration information to determine the address of the appropriate SOCKSv5 server for a target destination, the "best" match is used. The "best" match is defined as:

Item	Description
destination is numeric	Most bits in comparison (i.e. largest *prefixlength*)
destination is a name fragment	Most characters in name fragment.

When both name fragment and numeric addresses are present, all name fragment entries are "better" than numeric address entries.

Two implicit records:

```
0.0.0.0/0 NONE #All IPv4 destinations; no associated server.
```

```
::/0      NONE #All IPv6 destinations; no associated server.
```

are assumed as defaults for all destinations not specified in **/etc/socks5c.conf**.

Security

Access Control: This file should grant read (r) access to all users and grant write (w) access only to the root user.

Examples

```
#Sample socks5c.conf file
```

```
9.0.0.0/8    NONE    #Direct communication with all hosts in the 9 network.

129.35.0.0/16    sox1.austin.ibm.com

ibm.com    NONE    #Direct communication will all hosts matching "ibm.com" (e.g.
"aguila.austin.ibm.com")
```

Related reference
connect Subroutine

socks5tcp_accept Subroutine

Purpose

Awaits an incoming connection to a socket from a previous socks5tcp_bind() call.

Library

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

Syntax

```
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
```

```
int socks5tcp_accept(Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SrvLen;
```

Description

The **socks5tcp_accept** subroutine blocks until an incoming connection is established on a listening socket that was requested in a previous call to **socks5tcp_bind**. Upon success, subsequent writes to and reads from *Socket* will be relayed through *Svr*.

Socket must be an open socket descriptor of type SOCK_STREAM.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the unique name of the socket.
Dst	If non-NULL, buffer for receiving the address of the remote client which initiated an incoming connection
DstLength	Specifies the length of the address structure in *Dst*.
Svr	If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the server-side address of the incoming connection.
SvrLength	Specifies the length of the address structure in *Svr*.

Return Values

Upon successful completion, the **socks5tcp_accept** subroutine returns a value of 0, and modifies *Dst* and *Svr* to reflect the actual endpoints of the incoming external socket.

If the **socks5tcp_accept** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5_errno** global variable.

Error Codes (placed in errno; inherited from underlying call to connect())

The **socks5tcp_bindaccept** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
ENETUNREACH	No route to the network or host is present.
EFAULT	The *Dst* or *Svr* parameter is not in a writable part of the user address space.
EINVAL	One or more of the specified arguments is invalid.
ENETDOWN	The specified physical network is down.
ENOSPC	There is no space left on a device or system table.

Error	Description
ENOTCONN	The socket could not be connected.

Error Codes (placed in socks5_errno; SOCKSv5-specific errors)

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
S5_ESRVFAIL	General SOCKSv5 server failure.
S5_EPERM	SOCKSv5 server ruleset rejection.
S5_ENETUNREACH	SOCKSv5 server could not reach target network.
S5_EHOSTUNREACH	SOCKSv5 server could not reach target host.
S5_ECONNREFUSED	SOCKSv5 server connection request refused by target host.
S5_ETIMEDOUT	SOCKSv5 server connection failure due to TTL expiry.
S5_EOPNOTSUPP	Command not supported by SOCKSv5 server.
S5_EAFNOSUPPORT	Address family not supported by SOCKSv5 server.
S5_EADDRINUSE	Requested bind address is already in use (at the SOCKSv5 server).
S5_ENOSERV	No server found.

Examples

The following program fragment illustrates the use of the **socks5tcp_accept** and **socks5tcp_bind** subroutines by a client to request a listening socket from a server and wait for an incoming connection on the server side.

```
struct sockaddr_in svr;
struct sockaddr_in dst;
.
.
.
socks5tcp_bind(s,(struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr), &res,
sizeof(svr));
.
.
.
socks5tcp_accept(s, (struct sockaddr *)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

Related information

Initiating Stream Connections Example Program

Sockets Overview

socks5tcp_bind Subroutine

Purpose

Connect to a SOCKSv5 server and request a listening socket for incoming remote connections.

Library

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

Syntax

```
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
```

```
int socks5tcp_bind(Socket, Dst, DstLen, Svr, SvrLen)
Int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SrvLen;
```

Description

The **socks5tcp_bind** subroutine requests a listening socket on the SOCKSv5 server specified in *Svr*, in preparation for an incoming connection from a remote destination, specified by *Dst*. Upon success, *Svr* will be overwritten with the actual address of the newly bound listening socket, and *Socket* may be used in a subsequent call to **socks5tcp_accept**.

Socket must be an open socket descriptor of type SOCK_STREAM.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the unique name of the socket.
Dst	Specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the actual bound address on the server.
DstLength	Specifies the length of the address structure in *Dst*.
Svr	If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection; on success, this space will be overwritten with the actual bound address on the server.
SvrLength	Specifies the length of the address structure in *Svr*.

Return Values

Upon successful completion, the **socks5tcp_bind** subroutine returns a value of 0, and modifies *Svr* to reflect the actual address of the newly bound listener socket.

If the **socks5tcp_bind** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5_errno** global variable.

Error Codes (placed in errno; inherited from underlying call to connect())

The **socks5tcp_bindaccept** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EADDRNOTAVAIL	The specified address is not available from the local machine.

Error	Description
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EISCONN	The socket is already connected.
ETIMEDOUT	The establishment of a connection timed out before a connection was made.
ECONNREFUSED	The attempt to connect was rejected.
ENETUNREACH	No route to the network or host is present.
EADDRINUSE	The specified address is already in use.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINPROGRESS	The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.
EINVAL	One or more of the specified arguments is invalid.
ENETDOWN	The specified physical network is down.
ENOSPC	There is no space left on a device or system table.
ENOTCONN	The socket could not be connected.

Error Codes (placed in socks5_errno; SOCKSv5-specific errors)

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
S5_ESRVFAIL	General SOCKSv5 server failure.
S5_EPERM	SOCKSv5 server ruleset rejection.
S5_ENETUNREACH	SOCKSv5 server could not reach target network.
S5_EHOSTUNREACH	SOCKSv5 server could not reach target host.
S5_ECONNREFUSED	SOCKSv5 server connection request refused by target host.
S5_ETIMEDOUT	SOCKSv5 server connection failure due to TTL expiry.
S5_EOPNOTSUPP	Command not supported by SOCKSv5 server.
S5_EAFNOSUPPORT	Address family not supported by SOCKSv5 server.
S5_EADDRINUSE	Requested bind address is already in use (at the SOCKSv5 server).
S5_ENOERV	No server found.

Examples

The following program fragment illustrates the use of the **socks5tcp_bind** subroutine by a client to request a listening socket from a server.

```
struct sockaddr_in svr;
struct sockaddr_in dst;
.
.
.
socks5tcp_bind(s, (struct sockaddr *)&dst, sizeof(dst), (structsockaddr *)&svr, sizeof(svr));
```

Related reference

getsockname Subroutine

Related information
Sockets Overview

socks5tcp_connect Subroutine

Purpose

Connect to a SOCKSv5 server and request a connection to an external destination.

Library

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

Syntax

```
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
```

```
int socks5tcp_connect (Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SrvLen;
```

Description

The **socks5tcp_connect** subroutine requests a connection to *Dst* from the SOCKSv5 server specified in *Svr*. If successful, *Dst* and *Svr* will be overwritten with the actual addresses of the external connection and subsequent writes to and reads from *Socket* will be relayed through *Svr*.

Socket must be an open socket descriptor of type SOCK_STREAM; *Dst* and *Svr* may be either IPv4 or IPv6 addresses.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the unique name of the socket.
Dst	Specifies the external address of the target socket to which the SOCKSv5 server will attempt to connect.
DstLength	Specifies the length of the address structure in *Dst*.
Svr	If non-NULL, specifies the address of the SOCKSv5 server to use to request the relayed connection.
SvrLength	Specifies the length of the address structure in *Svr*.

Return Values

Upon successful completion, the **socks5tcp_connect** subroutine returns a value of 0, and modifies *Dst* and *Svr* to reflect the actual endpoints of the created external socket.

If the **socks5tcp_connect** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5_errno** global variable.
- *Dst* and *Svr* are left unmodified.

Error Codes (placed in errno; inherited from underlying call to connect())

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EISCONN	The socket is already connected.
ETIMEDOUT	The establishment of a connection timed out before a connection was made.
ECONNREFUSED	The attempt to connect was rejected.
ENETUNREACH	No route to the network or host is present.
EADDRINUSE	The specified address is already in use.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINPROGRESS	The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.
EINVAL	One or more of the specified arguments is invalid.
ENETDOWN	The specified physical network is down.
ENOSPC	There is no space left on a device or system table.
ENOTCONN	The socket could not be connected.

Error Codes (placed in socks5_errno; SOCKSv5-specific errors)

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
S5_ESRVFAIL	General SOCKSv5 server failure.
S5_EPERM	SOCKSv5 server ruleset rejection.
S5_ENETUNREACH	SOCKSv5 server could not reach target network.
S5_EHOSTUNREACH	SOCKSv5 server could not reach target host.
S5_ECONNREFUSED	SOCKSv5 server connection request refused by target host.
S5_ETIMEDOUT	SOCKSv5 server connection failure due to TTL expiry.
S5_EOPNOTSUPP	Command not supported by SOCKSv5 server.
S5_EAFNOSUPPORT	Address family not supported by SOCKSv5 server.
S5_ENOSERV	No server found.

Examples

The following program fragment illustrates the use of the **socks5tcp_connect** subroutine by a client to request a connection from a server's socket.

```
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
.
socks5tcp_connect(s,(struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

Related reference

getsockname Subroutine

Related information

Initiating Stream Connections Example Program

socks5udp_associate Subroutine

Purpose

Connects to a SOCKSv5 server, and requests a UDP association for subsequent UDP socket communications.

Library

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

Syntax

```
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
```

```
int socks5udp_associate (Socket, Dst, DstLen, Svr, SvrLen)
int Socket;
const struct sockaddr *Dst;
size_t DstLen;
const struct sockaddr *Svr;
size_t SrvLen;
```

Description

The **socks5udp_associate** subroutine requests a UDP association for *Dst* on the SOCKSv5 server specified in *Svr*. Upon success, *Dst* is overwritten with a rendezvous address to which subsequent UDP packets should be sent for relay by *Svr*.

Socket must be an open socket descriptor of type SOCK_STREAM; *Dst* and *Svr* may be either IPv4 or IPv6 addresses.

Note that *Socket* cannot be used to send subsequent UDP packets (a second socket of type SOCK_DGRAM must be created).

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the unique name of the socket.

Item	Description
Dst	Specifies the external address of the target socket to which the SOCKSv5 client expects to send UDP packets.
DstLength	Specifies the length of the address structure in *Dst*.
Svr	Specifies the address of the SOCKSv5 server to use to request the association.
SvrLength	Specifies the length of the address structure in *Svr*.

Return Values

Upon successful completion, the **socks5udp_associate** subroutine returns a value of 0 and overwrites *Dst* with the rendezvous address.

If the **socks5udp_associate** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5_errno** global variable.

Error Codes (placed in errno; inherited from underlying call to connect())

The **socks5udp_associate** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EADDRNOTAVAIL	The specified address is not available from the local machine.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
EISCONN	The socket is already connected.
ETIMEDOUT	The establishment of a connection timed out before a connection was made.
ECONNREFUSED	The attempt to connect was rejected.
ENETUNREACH	No route to the network or host is present.
EADDRINUSE	The specified address is already in use.
EFAULT	The *Address* parameter is not in a writable part of the user address space.
EINPROGRESS	The socket is marked as nonblocking. The connection cannot be immediately completed. The application program can select the socket for writing during the connection process.
EINVAL	One or more of the specified arguments is invalid.
ENETDOWN	The specified physical network is down.
ENOSPC	There is no space left on a device or system table.
ENOTCONN	The socket could not be connected.

Error Codes (placed in socks5_errno; SOCKSv5-specific errors)

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
S5_ESRVFAIL	General SOCKSv5 server failure.
S5_EPERM	SOCKSv5 server ruleset rejection.
S5_ENETUNREACH	SOCKSv5 server could not reach target network.
S5_EHOSTUNREACH	SOCKSv5 server could not reach target host.
S5_ECONNREFUSED	SOCKSv5 server connection request refused by target host.
S5_ETIMEDOUT	SOCKSv5 server connection failure due to TTL expiry.
S5_EOPNOTSUPP	Command not supported by SOCKSv5 server.
S5_EAFNOSUPPORT	Address family not supported by SOCKSv5 server.
S5_ENOSERV	No server found.

Examples

The following program fragment illustrates the use of the **socks5udp_associate** subroutine by a client to request an association on a server.

```
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
.
socks5udp_associate(s,(struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr,
sizeof(svr));
```

Related information

Initiating Stream Connections Example Program

SOCKS5C_CONFIG Environment Variable

socks5udp_sendto Subroutine

Purpose

Send UDP packets through a SOCKSv5 server.

Library

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

Syntax

```
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
```

```
int socks5udp_sendto (Socket, Message, MsgLen, Flags, Dst, DstLen, Svr, SvrLen)
int Socket;
void *Message;
size_t MsgLen;
int Flags;
struct sockaddr *Dst;
size_t DstLen;
struct sockaddr *Svr;
size_t SrvLen;
```

Description

The **socks5udp_sendto** subroutine sends a UDP packet to *Svr* for relay to *Dst*. *Svr* must be the rendezvous address returned from a previous call to **socks5udp_associate**.

Socket must be an open socket descriptor of type SOCK_DGRAM; *Dst* and *Svr* may be either IPv4 or IPv6 addresses.

The socket applications can be compiled with **COMPAT_43** defined. This will make the **sockaddr** structure BSD 4.3 compatible. For more details refer to **socket.h**.

Parameters

Item	Description
Socket	Specifies the unique name of the socket.
Message	Specifies the address containing the message to be sent.
MsgLen	Specifies the size of the message in bytes.
Flags	Allows the sender to control the message transmission. See the description in the **sendto** subroutine for more specific details.
Dst	Specifies the external address to which the SOCKSv5 server will attempt to relay the UDP packet.
DstLength	Specifies the length of the address structure in *Dst*.
Svr	Specifies the address of the SOCKSv5 server to send the UDP packet for relay.
SvrLength	Specifies the length of the address structure in *Svr*.

Return Values

Upon successful completion, the **socks5udp_sendto** subroutine returns a value of 0.

If the **socks5udp_sendto** subroutine is unsuccessful, the system handler performs the following functions:

- Returns a value of -1 to the calling program.
- Moves an error code, indicating the generic system error, into the **errno** global variable.
- Moves an error code, indicating the specific SOCKSv5 error, into the **socks5_errno** global variable.

Error Codes (placed in errno; inherited from underlying call to sendto())

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
EBADF	The *Socket* parameter is not valid.
ENOTSOCK	The *Socket* parameter refers to a file, not a socket.
EAFNOSUPPORT	The addresses in the specified address family cannot be used with this socket.
ENETUNREACH	No route to the network or host is present.
EINVAL	One or more of the specified arguments is invalid.
ENETDOWN	The specified physical network is down.
ENOSPC	There is no space left on a device or system table.

Error Codes (placed in socks5_errno; SOCKSv5-specific errors)

The **socks5tcp_connect** subroutine is unsuccessful if any of the following errors occurs:

Error	Description
S5_ESRVFAIL	General SOCKSv5 server failure.
S5_EPERM	SOCKSv5 server ruleset rejection.
S5_ENETUNREACH	SOCKSv5 server could not reach target network.
S5_EHOSTUNREACH	SOCKSv5 server could not reach target host.
S5_ECONNREFUSED	SOCKSv5 server connection request refused by target host.
S5_ETIMEDOUT	SOCKSv5 server connection failure due to TTL expiry.
S5_EOPNOTSUPP	Command not supported by SOCKSv5 server.
S5_EAFNOSUPPORT	Address family not supported by SOCKSv5 server.
S5_ENOSERV	No server found.

Examples

The following program fragment illustrates the use of the **socks5udp_sendto** subroutine by a client to request a connection from a server's socket.

```
void *message;
size_t msglen;
int flags;
struct sockaddr_in svr;
struct sockaddr_in6 dst;
.
.
.
socks5udp_associate(s,(struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
.
.
.
socks5udp_sendto(s, message, msglen, flags (struct sockaddr*)&dst, sizeof(dst), (struct sockaddr *)&svr, sizeof(svr));
```

Related information

Sockets Overview

splice Subroutine

Purpose

Lets the protocol stack manage two sockets that use TCP.

Syntax

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

int splice(*socket1, socket2, flags***)**
  int *socket1*, *socket2*;
  int *flags*;

Description

The **splice** subroutine will let TCP manage two sockets that are in connected state thus relieving the caller from moving data from one socket to another. After the **splice** subroutine returns successfully, the caller needs to close the two sockets.

The two sockets should be of type **SOCK_STREAM** and protocol **IPPROTO_TCP**. Specifying a protocol of zero will also work.

Parameters

Item	Description
socket1, socket2	Specifies a socket that had gone through a successful connect() or accept().
flags	Set to zero. Currently ignored.

Return Values

Item	Description
0	Indicates a successful completion.
-1	Indicates an error. The specific error is indicated by errno.

Error Codes

Item	Description
EBADF	*socket1* or *socket2* is not valid.
ENOTSOCK	*socket1* or *socket2* refers to a file, not a socket.
EOPNOTSUPP	*socket1* or *socket2* is not of type **SOCK_STREAM**.
EINVAL	The parameters are invalid.
EEXIST	*socket1* or *socket2* is already spliced.
ENOTCONN	*socket1* or *socket2* is not in connected state.
EAFNOSUPPORT	*socket1* or *socket2* address family is not supported for this subroutine.

WriteFile Subroutine

Purpose

Writes data to a socket.

Syntax

```
#include <iocp.h>
boolean_t WriteFile (FileDescriptor, Buffer, WriteCount, AmountWritten, Overlapped)
HANDLE FileDescriptor;
LPVOID Buffer;
DWORD WriteCount;
LPDWORD AmountWritten;
LPOVERLAPPED Overlapped;
```

Description

The **WriteFile** subroutine writes the number of bytes specified by the *WriteCount* parameter from the buffer indicated by the *Buffer* parameter to the *FileDescriptor* parameter. The number of bytes written is saved in the *AmountWritten* parameter. The *Overlapped* parameter indicates whether or not the operation can be handled asynchronously.

The **WriteFile** subroutine returns a boolean (an integer) indicating whether or not the request has been completed.

The **WriteFile** subroutine is part of the I/O Completion Port (IOCP) kernel extension.

Note: This subroutine only works to a socket file descriptor. It does not work with files or other file descriptors.

Parameters

Item	Description
FileDescriptor	Specifies a valid file descriptor obtained from a call to the **socket** or **accept** subroutines.
Buffer	Specifies the buffer from which the data will be written.
WriteCount	Specifies the maximum number of bytes to write.
AmountWritten	Specifies the number of bytes written. The parameter is set by the subroutine.
Overlapped	Specifies an overlapped structure indicating whether or not the request can be handled asynchronously.

Return Values

Upon successful completion, the **WriteFile** subroutine returns a boolean indicating the request has been completed.

If the **WriteFile** subroutine is unsuccessful, the subroutine handler performs the following functions:

- Returns a value of 0 to the calling program.
- Moves an error code, indicating the specific error, into the **errno** global variable. For further explanation of the **errno** variable, see the link in the Related Information section of this document.

Error Codes

Item	Description
EINPROGRESS	The write request can not be immediately satisfied and will be handled asynchronously. A completion packet will be sent to the associated completion port upon completion.
EAGAIN	The write request cannot be immediately satisfied and cannot be handled asynchronously.
EINVAL	The *FileDescriptor* is invalid.

Examples

The following program fragment illustrates the use of the **WriteFile** subroutine to synchronously write data to a socket:

```
void buffer;
int amount_written;
b=WriteFile (34, &buffer, 128, &amount_written, NULL);
```

The following program fragment illustrates the use of the **WriteFile** subroutine to asynchronously write data to a socket:

```
void buffer;
int amount_written;
LPOVERLAPPED overlapped;
b = ReadFile (34, &buffer, 128, &amount_written, overlapped);
```

Note: The request will only be handled asynchronously if it cannot be immediately satisfied.

Related information

Error Notification Object Class

Packet Capture

The packet capture library contains subroutines that allow users to communicate with the packet capture facility provided by the operating system to read unprocessed network traffic. Applications using these subroutines must be run as root. These subroutines are maintained in the **libpcap.a** library:

Related information

pcap_close

pcap_strerror

ioctl BPF Control Operations

Purpose

Performs packet-capture-related control operations.

Syntax

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

```
int ioctl ( int fd, int cmd[, arg ])
```

Description

The Berkeley Packet Filter (BPF) ioctl commands perform a variety of packet-capture-related control. The *fd* argument is a BPF device descriptor. For non-packet-capture descriptors, functions performed by this call are unspecified.

The *cmd* parameter and an optional third parameter (with varying types) are passed to and interpreted by the BPF ioctl function to perform an appropriate control operation that is specified by the user.

Parameters

Item	Description
fd	Specifies an open file descriptor that refers to a BPF device created using the open call.
cmd	Selects the control function to be performed.
arg	Represents additional information that is needed to perform the requested function. The type of the *arg* parameter is either an integer or a pointer to a BPF-specific data structure, depending on the particular control request.

BPF Control Operations

In addition to the **FIONREAD** ioctl command, the following commands can be applied to any open BPF device. The *arg* parameter is a pointer to the indicated type.

ioctl command	Type of the *arg* parameter	Description
BIOCGBLEN	**u_int**	Returns the buffer length for reads on BPF devices.

ioctl command	Type of the *arg* parameter	Description
BIOCSBLEN	**u_int**	Sets the buffer length for reads on BPF devices. The *buffer* parameter must be set before the device is attached to an interface with the **BIOCSETIF** command. If the requested buffer size cannot be accommodated, the closest allowable size is set and returned in the *arg* parameter.
BIOCGDLT	**u_int**	Returns the type of the data link layer underlying the attached interface.
BIOCPROMISC	N/A	Forces the interface into promiscuous mode. All packets, not just those destined for the local host, are processed. A listener that opened its interface nonpromiscuously can receive packets promiscuously, because more than one device can be listening on a given interface. The problem can be remedied with an appropriate filter.
BIOCFLUSH	N/A	Flushes the buffer of incoming packets, and resets the statistics that are returned by the **BIOCGSTATS** command.
BIOCGETIF	**struct ifreq**	Returns the name of the hardware interface that the device is listening on. The name is returned in the **ifr_name** field of the **ifreq** structure. All other fields are undefined.
BIOCSETIF	**struct ifreq**	Sets the hardware interface associate with the device. This command must be performed before any pack-packets can be read. The device is indicated by the name using the **ifr_name** field of the **ifreq** structure. Additionally, the command performs the actions of the **BIOCFLUSH** command.
BIOCGRTIMEOUT	**struct timeval**	Gets the read timeout value. The *arg* parameter specifies the length of time to wait before a read request times out. This parameter is initialized to zero by an open, indicating no timeout.
BIOCSRTIMEOUT	**struct timeval**	Sets the read timeout value described in the **BIOCGRTIMEOUT** command.
BIOCGSTATS	**struct bpf_stat**	Returns the a structure of packet statistics. The structure is defined in the **net/bpf.h** file.
BIOCIMMEDIATE	**u_int**	Enables or disables the **immediate** mode, based on the truth value of the *arg* parameter. When the **immediate** mode is enabled, reads return immediately upon packet reception. Otherwise, a read will be blocked until either the kernel buffer becomes full or a timeout occurs.
BIOCSETF	**struct bpf_program**	Sets the filter program used by the kernel to discard uninteresting packets. The **bpf_program** structure is defined in the **net/bpf.h** file.

ioctl command	Type of the *arg* parameter	Description
BIOCVERSION	**struct bpf_version**	Returns the major and minor version numbers of the filter language currently recognized by the kernel. Before installing a filter, applications must check that the current version is compatible with the running kernel. The current version numbers are given by the BPF_MAJOR_VERSION and BPF_MINOR_VERSION variables from the **net/bpf.h** file. An incompatible filter might result in undefined behavior.

Return Values

Upon successful completion, ioctl returns a value of `0`. Otherwise, it returns a value of `-1` and sets `errno` to indicate the error.

Error Codes

The ioctl commands fail under the following general conditions:

Item	Description
EINVAL	A command or argument, which is not valid, was specified.
ENETDOWN	The underlying interface or network is down.
ENXIO	The underlying interface is not found.
ENOBUFS	Insufficient memory was available to process the request.
EEXIST	The BPF device already exists.
ENODEV	The BPF device could not be set up.
EINTR	A signal was caught during an ioctl operation.
EACCES	The permission was denied for the specified operation.
EADDRNOTAVAIL	The specified address is not available for interface.
ENOMEM	The available memory is not enough.
ESRCH	Such a process does not exist.

Related information

Packet Capture Library Overview

Librdmacm Library

The librdmacm library provides the connection management (CM) functionality and the CM interfaces for remote direct memory access (RDMA).

The API user space is described in the `/usr/include/rdma/rdma_cma.h` file.

The manual pages are created to describe the various interfaces and test programs that are available. For a full list of interfaces and test programs, refer to the `rdma_cm` manual page.

Returned error rules

The **librdmacm** functions return 0 to indicate success, and a negative value to indicate failure.

If a function operates asynchronously, a return value of 0 means that the operation was successfully started. The operation might still return an error. You must check the status of the related event. If the return value is -1, the **errno** can be examined for additional information of the failure.

Item	Description
=0	Success
= -1	Error . See the **errno** for details of the error message.

Supported verbs

You can find a list of verbs supported by the librdmacm library.

Event channel operations

Lists the event channel operations that are handled for the library verbs.

rdma_create_event_channel

Opens a channel that is used to report communication events.

Syntax

```
#include <rdma/rdma_cma.h>
struct rdma_event_channel *rdma_create_event_channel(void);
```

Description

The **rdma_create_event_channel** function reports the asynchronous events through event channels. Each event channel maps to a file descriptor.

Note:

- Event channels are used to direct all events on an **rdma_cm_id** identifier. You might require multiple event channels when you are managing a large number of connections or connection manager (CM) ID's.
- All event channels that are created must be destroyed by calling the **rdma_destroy_event_channel** function. You must call the **rdma_get_cm_event** function to retrieve events on an event channel.

Parameters

Item	Description
void	No arguments.

Return Value

The **rdma_create_event_channel** function returns 0 on success, and NULL if the request fails. On failure, **errno** indicates the reason for failure.

rdma_destroy_event_channel

Closes an event communication channel.

Syntax

```
#include <rdma/rdma_cma.h>
void rdma_destroy_event_channel(struct rdma_event_channel *channel);
```

Description

The **rdma_destroy_event_channel** function releases all resources that are associated with an event channel and closes the associated file descriptor.

Note: The `rdma_cm_id` identifiers that are associated with the event channel must be destroyed, and all returned events must be acknowledged before calling the **rdma_destroy_event_channel** function.

Parameters

Item	Description
channel	Specifies the communication channel to be destroyed.

Return Value

The **rdma_destroy_event_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

Connection Manager (CM) ID operations

The Connection Manager (CM) ID operation is used for ID related operations such as to create, destroy, migrate, resolve address, establish connection, listen to the request, reject request, and to provide the address information.

rdma_cm

Establishes communication over RDMA transports.

Syntax

```
#include <rdma/rdma_cma.h>
```

Description

Establishes communication over RDMA transports.

Notes:

- The RDMA CM is a communication manager (CM) used to set up reliable, connected, and unreliable datagram data transfers. It provides an RDMA transport neutral interface for establishing connections. The API concepts are based on sockets, but adapted for queue pair (QP) based semantics. The communication for QP must be over a specific RDMA device, and data transfers are message-based.
- The RDMA CM can control both the QP and communication management (that is connection setup or teardown) functions of an RDMA API, or only the communication management. It works in conjunction with the verbs API that is defined by the libibverbs library. The libibverbs library provides the underlying interfaces needed to send and receive data.
- The RDMA CM can operate asynchronously or synchronously. The mode of operation is controlled by using the **rdma_cm** event channel parameter in specific calls. If an event channel is provided, an **rdma_cm** identifier reports its event data (that is results of establishing a connection, for example), on that channel. If a channel is not provided, then all **rdma_cm** operation for the selected **rdma_cm** identifier is blocked until the channel completes.

RDMA verbs

The **rdma_cm** manager supports the verbs that are available in the libibverbs library and interfaces. However, it also provides wrapper functions for the commonly used verbs. The set of abstracted verb call are:

rdma_reg_msgs
Registers an array of buffers for sending and receiving.

rdma_reg_read
Registers a buffer for RDMA read operations.

rdma_reg_write
Registers a buffer for RDMA write operations.

rdma_dereg_m
Reregisters a memory region.

rdma_post_recv
Posts a buffer to receive a message.

rdma_post_send
Posts a buffer to send a message.

rdma_post_read
Posts an RDMA to read data into a buffer.

rdma_post_write
Posts an RDMA to send data from a buffer.

rdma_post_recvv
Posts a vector of buffers to receive a message.

rdma_post_sendv
Posts a vector of buffers to send a message.

rdma_post_readv
Posts a vector of buffers to receive an RDMA read.

rdma_post_writev
Posts a vector of buffers to send an RDMA write.

rdma_post_ud_send
Posts a buffer to send a message on a UD QP.

rdma_get_send_comp
Gets completion status for a send or RDMA operation.

rdma_get_recv_comp
Gets information about a completed receive.

Examples

1. CLIENT operation

   An overview of the basic operation for the active, or client, side of communication is described in this section. This flow is for asynchronous operation with low-level call details. For synchronous operation, calls to **rdma_create_event_channel**, **rdma_get_cm_event**, **rdma_ack_cm_event**, and **rdma_destroy_event_channel** is eliminated. Abstracted calls, such as **rdma_create_ep** contains several calls under a single API. A general connection flow includes the following calls:

   rdma_getaddrinfo
   Retrieves address information of the destination.

   rdma_create_event_channel
   Creates channel to receive events.

   rdma_create_id
   Allocates an **rdma_cm_id** identifier, this call is similar in function to a socket.

   rdma_resolve_addr
   Obtains a local RDMA device to reach the remote address.

   rdma_get_cm_event
   Waits for `RDMA_CM_EVENT_ADDR_RESOLVED` event.

   rdma_ack_cm_event
   Acknowledges an event.

   rdma_create_qp
   Allocates a queue pair (QP) for the communication.

   rdma_resolve_route
   Determines the route to the remote address.

rdma_get_cm_event
Waits for theRDMA_CM_EVENT_ROUTE_RESOLVED event.

rdma_ack_cm_event
Acknowledges an event.

rdma_connect
Connects to the remote server.

rdma_get_cm_event
Waits for the RDMA_CM_EVENT_ESTABLISHED event

rdma_ack_cm_event
Acknowledges an event.

To perform data transfers over connection, follow these steps:

rdma_disconnect
Tears-down a connection.

rdma_get_cm_event
Waits for an RDMA_CM_EVENT_DISCONNECTED event.

rdma_ack_cm_event
Acknowledges an event.

rdma_destroy_qp
Destroys the QP.

rdma_destroy_id
Releases the **rdma_cm_id** identifier.

rdma_destroy_event_channel
Releases the event channel.

An identical process is used to set up unreliable datagram (UD) communication between nodes. No actual connection is formed between the queue pairs, so disconnection is not required. This example shows initiating the client for disconnect, either side of a connection can initiate the disconnect.

2. Server connection

A general overview of the basic operation for the passive, or server, side of communication is explained. A general connection flow includes the following events:

rdma_create_event_channel
Creates channel to receive events.

rdma_create_id
Allocates an **rdma_cm_id** identifier, this call is similar in function to a socket.

rdma_bind_addr
Sets the local port number to listen.

rdma_listen
Begins to listen for connection requests.

rdma_get_cm_event
Waits for RDMA_CM_EVENT_CONNECT_REQUEST event with a new **rdma_cm_id** identifier.

rdma_create_qp
Allocates a QP for the communication on the new **rdma_cm_id** identifier.

rdma_accept
Accepts the connection request.

rdma_ack_cm_event
Acknowledges an event.

rdma_get_cm_event
Waits for RDMA_CM_EVENT_ESTABLISHED event.

rdma_ack_cm_event
Acknowledges an event.

To perform data transfers over connection, follow these steps:

rdma_get_cm_event
Waits for an RDMA_CM_EVENT_DISCONNECTED event.

rdma_ack_cm_event
Acknowledges an event.

rdma_disconnect
Tears-down a connection.

rdma_destroy_qp
Destroys the QP.

rdma_destroy_id
Releases the connected**rdma_cm_id** identifier.

rdma_destroy_id
Releases the listening **rdma_cm_id** identifier.

rdma_destroy_event_channel
Releases the event channel.

Exit Status

= 0
Success

= -1
Error. See **errno** for more details.

Most librdmacm functions return 0 to indicate success, and a -1 return value to indicate failure. If a function operates asynchronously, a return value of 0 means that the operation started successfully. The operation can complete in error, and you must check the status of the related event. If the return value is -1, then **errno** contains additional information for the failure.

Note: The earlier versions of the library would return **-errno** and is not set to **errno** for some cases related to **ENOMEM**, **ENODEV**, **ENODATA**, **EINVAL**, and **EADDRNOTAVAIL** codes. Applications that require to verify the earlier version of the codes and that are compatible must manually set **errno** to negative of the return code, if it is **< -1**.

rdma_create_id

Allocates a communication identifier.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_create_id(struct rdma_event_channel *channel, struct rdma_cm_id **id, void *context,
enum rdma_port_space ps);
```

Description

The **rdma_create_id** function creates an identifier that is used to track communication information. The communication channel that the events are associated with the allocated **rdma_cm_id** identifier is communicated. This may be NULL.

Notes:

- The rdma_cm_id identifiers are equivalent to that of a socket in RDMA communication. The difference is that the RDMA communication requires explicit binding to a specified Remote Direct Memory Access (RDMA) device before communicating, and most operations are asynchronous in nature. The asynchronous communication events on an **rdma_cm_id** identifier are reported through the associated

event channel. If the channel parameter is NULL, the **rdma_cm_id** is placed into synchronous operation. While operating synchronously, calls that result in an event cause a block until the operation completes. The event is returned to the user through the **rdma_cm_id** structure, and is available for access until the next **rdma_cm** call is made.

- You must release the `rdma_cm_id` identifier by calling the **rdma_destroy_id** function.

Port Spaces: RDMA_PS_TCP provides reliable, connection-oriented queue pair (QP). Unlike TCP, the RDMA port space provides stream-based communication.

Parameters

Item	Description
channel	Specifies the communication channel for the allocated `rdma_cm_id` identifier to report the associated events.
context	Indicates the user-specified context that is associated with the communication identifier.
id	Specifies a reference identifier to return the allocated communication identifier.
ps	Specifies the RDMA port space.

Return Values

The **rdma_destroy_event_channel** function returns 0 on success, or -1 on error. If an error occurs, the **errno** indicates the reason for failure.

rdma_destroy_id

Releases a communication identifier.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_destroy_id(struct rdma_cm_id *id);
```

Description

The **rdma_destroy_id** function destroys the specified `rdma_cm_id` identifier and cancels any outstanding asynchronous operation.

Note: You must release any queue pair (QP) that is associated with the `rdma_cm_id` identifier before you call the **rdma_destroy_id** function and acknowledge all the related events.

Parameters

Item	Description
id	Specifies the communication identifier to destroy.

Return Values

The **rdma_destroy_event_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

rdma_migrate_id

Moves a communication identifier to another event channel.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_migrate_id(struct rdma_cm_id *id, struct rdma_event_channel *channel);
```

Description

The **rdma_migrate_id** function migrates a communication identifier to a different event channel and moves the pending events associated with the `rdma_cm_id` identifier to the new channel.

Notes:

- You must not poll for current event channel on the `rdma_cm_id` identifiers or run any other routines on the`rdma_cm_id` identifier when migrating between channels.
- The **rdma_migrate_id** operation stops if any unacknowledged events are on the current event channel.
- If the channel parameter is NULL, the specified `rdma_cm_id` identifier is placed into synchronous operation mode. All calls on the ID is blocked until the operation completes.

Parameters

Item	Description
id	Specifies the existing communication identifier to migrate.
channel	Specifies the communication channel that the events associated with the allocated `rdma_cm_id` identifier reports. This parameter may be NULL.

Return Values

The **rdma_migrate_id** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

rdma_bind_addr

Binds an remote direct memory access (RDMA) identifier to a source address.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_bind_addr(struct rdma_cm_id *id, struct sockaddr *addr);
```

Description

The **rdma_bind_addr** function associates a source address with an `rdma_cm_id` identifier. The address can be a wildcard value. If an `rdma_cm_id` identifier has a local address, the identifier also has a local RDMA device.

Notes:

- The **rdma_bind_addr** operation is run before the **rdma_listen** operation to bind to a specific port number. The **rdma_bind_addr** operation is run on the active side of a connection before the **rdma_resolve_addr** routine runs to bind to a specific address.
- If the **rdma_bind_addr** operation is used to bind to port 0, the `rdma_cm` function selects an available port that can be retrieved with the **rdma_get_src_port** operation.

Parameters

Item	Description
id	Specifies the RDMA identifier.
addr	Specifies the local address information. Wildcard values are permitted.

Return Values

The **rdma_bind_addr** function returns the following values:

Item	Description
0	On success.
-1	Error, see **errno**.

rdma_resolve_addr

Resolves the destination and optional source addresses.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_resolve_addr(struct rdma_cm_id *id, struct sockaddr *src_addr, struct sockaddr
*dst_addr,
 int timeout_ms);
```

Description

The **rdma_resolve_addr** function resolves the destination and optional source addresses from an IP address to an Remote Direct Memory Access (RDMA) address. If successful, the specified `rdma_cm_id` identifier is associated with a local device.

Notes:

- The **rdma_resolve_addr** operation is used to map a specified destination IP address to a usable RDMA address. The IP- RDMA address mapping is done by using the local routing table, or by using ARP.
- If the source address is specified, the `rdma_cm_id` identifier is associated with the source address, and the situation is similar to running the **rdma_bind_addr** operation. If no source address is specified, the `rdma_cm_id` identifier is not associated with a device, and the identifier gets associated with a source address based on the local routing tables.
- The **rdma_resolve_addr** operation is run from the active side of a connection, before running the **rdma_resolve_route** and **rdma_connect** operations.

Parameters

Item	Description
id	Specifies the RDMA identifier.
src_addr	Specifies the source address information, and this parameter can be NULL.
dst_addr	Specifies the destination address information.
timeout_ms	Specifies the time of resolution.

Return Values

The **rdma_resolve_addr** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

rdma_resolve_route
Resolves the route information that is required to establish a connection.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_resolve_route(struct rdma_cm_id *id, int timeout_ms);
```

Description

The **rdma_resolve_route** function resolves an RDMA route to the destination address to establish a connection. The destination address must be resolved by running the **rdma_resolve_addr** subroutine.

Note: The **rdma_resolve_route** operation is called on the client side of a connection after running the **rdma_resolve_addr** operation, but before the **rdma_connect** operation.

Parameters

Item	Description
id	Specifies the RDMA identifier.
timeout_ms	Specifies the time of resolution.

Return Values

The **rdma_resolve_route** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

rdma_connect
Initiates an active connection request.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_connect(struct rdma_cm_id *id, struct rdma_conn_param *conn_param);
```

Description

The **rdma_connect** function initiates a connection request to a remote destination.

Note: The route to the destination address must be resolved by running the **rdma_resolve_route** call or by running the **rdma_create_ep** call before the **rdma_connect** operation.

Connection Properties

The following properties are used to configure the communication that is specified by the *conn_param* parameter when connecting or establishing a datagram communication.

private_data
References a user-controlled data buffer. The contents of the buffer are copied and transparently passed to the remote side as part of the communication request. This property can be NULL if it is not required.

private_data_len:
Specifies the size of the user-controlled data buffer.

responder_resources:
Specifies the maximum number of outstanding Remote Direct Memory Access (RDMA) read operations that the local side accepts from the remote side. This property applies only to the

RDMA_PS_TCP event. The **responder_resources** value must be less than or equal to the local RDMA device attribute **max_qp_rd_atom** and to the remote RDMA device attribute **max_qp_init_rd_atom**. The remote endpoint can adjust this value when accepting the connection.

initiator_depth:
Specifies the maximum number of outstanding RDMA read operations that the local side must read to the remote side. This property applies only to the RDMA_PS_TCP event. The **initiator_depth** value must be less than or equal to the local RDMA device attribute **max_qp_init_rd_atom** and to the remote RDMA device attribute **max_qp_rd_atom**. The remote endpoint can adjust to this value when accepting the connection.

flow_control:
Specifies if the hardware flow control is available. The **flow_control** value is exchanged with the remote peer and is not used to configure the queue pair (QP). This property applies only to the RDMA_PS_TCP event , and is specific to the InfiniBand architecture.

retry_count:
Specifies the maximum number of times the data transfer operation must be tried on the connection when an error occurs. The **retry_count** setting controls the number of times to retry sending the data transmission to RDMA, and atomic operations when time outs occur. This property applies only to the RDMA_PS_TCP event, and is specific to the InfiniBand architecture.

rnr_retry_count:
Specifies the maximum number of times that a send operation from the remote peer is tried on a connection after receiving a **receiver not ready** (RNR) error. RNR errors are generated when a send request arrives before a buffer is posted to receive the incoming data. This property applies only to the RDMA_PS_TCP event., and is specific to the InfiniBand architecture.

srq:
Specifies whether the QP that is associated with the connection is using a shared receive queue. The **srq** field is ignored by the library if a QP is created on the `rdma_cm_id` identifier. This property applies only to the RDMA_PS_TCP event, and is currently not supported.

qp_num:
Specifies whether the QP number is associated with the connection. The **qp_num** field is ignored by the library if a QP is created on the `rdma_cm_id` identifier. This property applies only to the RDMA_PS_TCP event.

iWARP specific:
Specifies the connections established over Internet Wide Area RDMA Protocol (iWARP RDMA) devices that currently require the active side of the connection to send the first message.

Parameters

Item	Description
id	Specifies the RDMA identifier.
conn_param	Specifies the connection parameters.

Return Values

The **rdma_connect** function returns the following values:

Item	Description
0	On success.
-1	Error, see **errno**.

rdma_listen
Listens to the incoming connection requests.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_listen(struct rdma_cm_id *id, int backlog);
```

Description

The **rdma_listen** function initiates a listen operation for the incoming connection requests. The listen operation is restricted to the locally bound source addresses.

Notes:

- You must have bound and associated the `rdma_cm_id` identifier with a local address by the **rdma_bind_addr** operation before the **rdma_listen** operation.
- If the `rdma_cm_id` identifier is bound to a specific IP address, the listen operation is restricted to that address and the associated RDMA device.
- If the `rdma_cm_id` identifier is bound to an RDMA port number, the listen operation occurs across all RDMA devices.

Parameters

Item	Description
id	Specifies the RDMA identifier.
backlog	Specifies the backlog of incoming connection requests.

Return Values

The **rdma_listen** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

rdma_accept
Accepts a connection request.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_accept(struct rdma_cm_id *id, struct rdma_conn_param *conn_param);
```

Description

The **rdma_accept** function is used to accept a connection lookup request.

Notes:

- The **rdma_accept** operation is not called on a listening `rdma_cm_id` identifier. After the **rdma_listen** operation is run, you must wait for a connection request event to occur.
- The `rdma_cm_id` identifier is created by the connection request events similar to a new socket, but the `rdma_cm_id` identifier is associated to a specific RDMA device. The **rdma_accept** operation is called on the new `rdma_cm_id` identifier.

Connection Properties

Refer to the **rdma_connect** routine for details on establishing a connection with the identifier.

Parameters

Item	Description
id	Specifies the connection identifier that is associated with the request.
conn_param	Specifies the information required to establish the connection.

Return Values

The **rdma_accept** function returns the following values:

Item	Description
0	On success.
-1	Error, see **errno**.

InfiniBand specific

The InfiniBand QPs are configured with minimum RNR NAK timer and local ACK timeout values. The minimum RNR NAK timer value is set to 0, for a delay of 655 ms. The local ACK timeout is calculated based on the packet lifetime and local HCA ACK delay. The packet lifetime is determined by the InfiniBand Subnet Administrator and is part of the route (path record) information that is obtained from the active side of the connection. The HCA ACK delay is a property of the locally used HCA.

The RNR retry count is a 3-bit value.

rdma_reject

Rejects a connection request.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_reject(struct rdma_cm_id *id, const void *private_data, uint8_t private_data_len);
```

Description

The **rdma_reject** function is run from the listening side of the connection to reject a connection lookup request.

Note: You can run the **rdma_reject** operation after receiving a connection request event. If the underlying RDMA transport function supports private data in the rejection message, the specified data is passed to the remote side.

Parameters

Item	Description
id	Specifies the connection identifier that is associated with the request.
private_data	Specifies the optional private data to send with the rejection message.
private_data_len	Specifies the size of the *private_data* parameter that can be sent, in bytes.

Return Values

The **rdma_reject** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

rdma_disconnect
Disconnects a connection.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_disconnect(struct rdma_cm_id *id);
```

Description

The **rdma_disconnect** function disconnects a connection and transitions any associated queue pair (QP) with the error state. The error state moves the work requests that are posted to the completion queue. This routing can be run by the client and server side of a connection. An RDMA_CM_EVENT_DISCONNECTED event is generated on both sides of the connection after successful disconnection.

Parameters

Item	Description
id	Specifies the connection identifier that is associated with the request.

Return Values

The **rdma_destroy_event_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

rdma_get_src_port
Returns the local port number of the associated communication identifier.

Syntax

```
#include <rdma/rdma_cma.h>
uint16_t rdma_get_src_port(struct rdma_cm_id *id)
```

Description

The **rdma_get_src_port** function returns the local port number for an `rdma_cm_id` identifier that is associated with a local address.

Parameters

Item	Description
id	Specifies the connection identifier that is associated with the request.

Return Values

The **rdma_get_src_port** function returns the 16-bit port identifier associated with the local endpoint. If the **rdma_cm_id** is not bound to a port, the returned value is 0.

rdma_get_dst_port
Returns the remote port number of the associated identifier.

Syntax

```
#include <rdma/rdma_cma.h>
uint16_t rdma_get_dst_port(struct rdma_cm_id *id)
```

Description

The **rdma_get_dst_port** function returns the remote port number for an `rdma_cm_id` identifier that is associated with a remote address.

Parameters

Item	Description
id	Specifies the connection identifier that is associated with the request.

Return Values

The **rdma_get_dst_port** function returns the 16-bit port identifier associated with the peer endpoint. If the `rdma_cm_id` is not connected, the returned value is 0.

rdma_get_local_addr
Returns the local IP address of the associated identifier.

Syntax

```
#include <rdma/rdma_cma.h>
struct sockaddr *rdma_get_local_addr(struct rdma_cm_id *id)
```

Description

The **rdma_get_local_addr** function returns the local IP address for an `rdma_cm_id` identifier that is associated with a local device.

Parameters

Item	Description
id	Specifies the RDMA identifier.

Return Values

The **rdma_get_local_addr** function returns the local IP address for an `rdma_cm_id` identifier that has been bound with a local device.

rdma_get_peer_addr
Returns the remote IP address of the associated communication identifier.

Syntax

```
#include <rdma/rdma_cma.h>
struct sockaddr *rdma_get_peer_addr(struct rdma_cm_id *id)
```

Description

The **rdma_get_peer_addr** function returns the remote IP address that is associated with an `rdma_cm_id` identifier.

Parameters

Item	Description
id	Specifies the RDMA identifier.

Return Values

The **rdma_get_peer_addr** function returns a pointer to the `sockaddr` address of the connected peer. If the `rdma_cm_id` identifier is not connected, the contents of the `sockaddr` structure is set to zero.

rdma_create_ep

Creates an identifier (**rdma_cm_id**) to track information about communication.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_create_ep [struct rdma_cm_id **id,struct rdma_addrinfo *res,
struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr,];
```

Description

The **rdma_cm_id** identifier allocates a communication identifier and an optional queue pair (QP). The **rdma_cm_id** identifier is used in one of the following methods:

- If the **rdma_cm_id** identifier is used on the active side of a connection, the `RAI_PASSIVE` option is not set on the `res->ai_flag` flag. The **rdma_create_ep** function automatically creates a QP on the **rdma_cm_id** identifier if the *qp_init_attr* value is not NULL. If the domain is provided, the QP is associated with the specified protection domain; otherwise, a default protection domain is used. After calling the **rdma_create_ep** function, the **rdma_cm_id** identifier that is returned can be connected by calling the **rdma_connect** function. The active side calls the **rdma_resolve_addr** function, and the **rdma_resolve_route** function is not necessary.
- If the **rdma_cm_id** identifer is used on the passive side of a connection, the `RAI_PASSIVE` option is set on the `res->ai_flag` flag. This function call saves the value of the *pd* and *qp_init_attr* parameters. A new connection request is retrieved by calling the **rdma_get_request** function. The **rdma_cm_id** identifier associated with the new connection is automatically associated with a QP by using the *pd* and *qp_init_attr* parameters. After calling the **rdma_create_ep** function, the **rdma_cm_id** identifier can be placed into a listening state by calling the **rdma_listen** function. The passive side call the **rdma_bind_addr** is not necessary. The **rdma_get_request** function can be used to retrieve the connection. The **rdma_cm_id** identifier that is created is used for synchronous operation. To choose the asynchronous operation you must move the **rdma_cm_id** identifier to a user-created event channel by using the **rdma_migrate_id** function.

Parameters

Item	Description
id	Specifies a reference by which the allocated communication identifier must be returned .
res	Specifies the address information that is associated with the **rdma_cm_id** identifier that is returned from the **rdma_getaddrinfo** function.

Item	Description
pd	Specifies the optional protection domain if a QP is associated with the **rdma_cm_id** identifier.
qp_init_attr	Specifies the optional initial, QP attributes.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_destroy_ep

Destroys the specified communication identifier and its associated resources.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_destroy_ep (struct rdma_cm_id *id)
```

Description

The **rdma_destroy_ep** function destroys the specified **rdma_cm_id** identifier and all its associated resources. The **rdma_destroy_ep** function automatically destroys any queue pair (QP) that is associated with the **rdma_cm_id** identifier.

Parameters

Item	Description
id	Specifies the communication identifier to destroy.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

rdma_getaddrinfo

Translates the transport independent address to establish communication.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_getaddrinfo (char *node, char *service, struct rdma_addrinfo *hints, struct
rdma_addrinfo **res);
```

Description

The **rdma_getaddrinfo** function resolves the destination node and service address and returns information that is required to establish communication. The function provides the RDMA functional equivalent to `getaddrinfo`.

Notes:

You must specify either node or service parameters for the translation. If hints are provided, the operation is controlled by the `hints.ai_flags` flag. If the `RAI_PASSIVE` flag is specified, the call resolves the address information that is used on the passive side of a connection.

Item	Description
ai_flags	Specifies the hint flags that control the operation. The following flags are supported: • RAI_PASSIVE: Indicates that the results are used on the passive or listening side of a connection. • RAI_NUMERICHOST: Indicates that if the flag is specified and if the node parameter is provided, the network address must be a numerical value. This flag suppresses any lengthy address resolution. • RAI_NOROUTE: Indicates that if the flag is set, the flag suppresses any lengthy route resolution.
ai_family	Specifies the address family for the source and destination address. The supported families are AF_IB, AF_INET, and AF_INET6.
ai_qp_type	Indicates the type of RDMA QP used for communication. The types that are supported are IBV_UD (unreliable datagram) and IBV_RC (reliable connected).
ai_port_space	Indicates the RDMA port space that is in use. The supported values are RDMA_PS_UDP and RDMA_PS_TCP.
ai_src_len	Indicates the length of the source address that is referenced by the ai_src_addr flag. If an appropriate source address for a given destination is not discovered the value of the ai_src_len flag is 0.
ai_dst_len	Indicates the length of the destination address that is referenced by ai_src_addr flag. This flag is set to 0, if the RAI_PASSIVE flag was specified as part of the hints.
ai_src_addr	Specifies the address for the local RDMA device, if the RDMA device is provided.
ai_dst_addr	Specifies the destination address for the RDMA device, if the RDMA device is provided.
ai_src_canonname	Specifies the canonical for the source.
ai_dst_canonname	Specifies the canonical for the destination.
ai_route_len	Specifies the size of the routing information buffer that is referenced by the ai_route flag. If the transport does not require routing data or none of the address could be resolved, the ai_route flag is 0.
ai_connect_len	Specifies the routing information for RDMA transports that require routing data for establishing the connection. The format of the routing data depends on the underlying transport. If InfiniBand transports are used, the ai_route flag references an array of ibv_path_data structures.
ai_connect	Specifies the size of connection information referenced by ai_route flag. If the underlying transport does not require any additional information to establish connection, the ai_connect flag is set to 0.
ai_next	Specifies the pointer to the next rdma_addrinfo structure in the list. The ai_next flag is NULL if no structures exist.

Parameters

Item	Description
hints	Specifies a reference to a `rdma_addrinfo` structure containing hints about the type of service the caller supports.
node	Specifies the optional name, dotted-decimal IPv4 or IPv6 hexadecimal address that must be resolved.
res	Specifies a pointer to a linked list of `rdma_addrinfo` structures that contains the response information.
service	Specifies the service name or port number of the address.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

rdma_notify

Notifies the asynchronous events that occurred on a queue pair (QP).

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_notify (struct rdma_cm_id *id, enum ibv_event_type event);
```

Description

Notifies the librdmacm of asynchronous events that occurred on a queue pair (QP) associated with the `rdma_cm_id` identifier.

Note: Asynchronous events that occur on a QP are reported through the device of the user event handler. This routine is used to notify the librdmacm of communication events. In most cases, use of this routine is not necessary. However if connection establishment is done out of band (such as InfiniBand), it is possible to receive data on a QP that is not yet considered connected. This routine force the connection into an established state in order to handle situations where the connection never forms on its own. Calling this routine ensures the delivery of the **RDMA_CM_EVENT_ESTABLISHED** event to the application. Events to be reported to the communication manager (CM) are **IB_EVENT_COMM_EST**.

Parameters

id
RDMA identifier.

event
Asynchronous event.

Exit Status

= 0
Success.

= -1
Error. See **errno** for more details on the error.

Event Handling Operations

Lists the event handling operations for the library verbs such as to get an event channel, acknowledge an event channel, and providing a string representation of the event channel.

rdma_get_cm_event

Retrieves the next pending communication event.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_get_cm_event(struct rdma_event_channel *channel, struct rdma_cm_event **event);
```

Description

The **rdma_get_cm_event** function retrieves a communication event. If no events are pending, the call is blocked until an event is received.

Notes:

- You can change the file descriptor that is associated with the channel and change the default synchronous behavior of the **rdma_get_cm_event** operation.
- All events that are reported must be acknowledged by running the **rdma_ack_cm_event** operation.
- The `rdma_cm_id` identifier is not destroyed until the related events are acknowledged.

Parameters

Item	Description
channel	Specifies the event channel to check for events.
event	Specifies the allocated information about the next communication event.

Return Values

Item	Description
0	On success.
-1	Error, see **errno**. If an error occurs, the **errno** indicates the reason for failure.

Event Data

The details of the communication event are returned to the **rdma_cm_event** function. This structure is allocated by the `rdma_cm` identifier and released by the **rdma_ack_cm_event** operation. The **rdma_cm_event** function has the following parameters:

Item	Description
id	Specifies the `rdma_cm` identifier that is associated with the event. If `RDMA_CM_EVENT_CONNECT_REQUEST` is the event type, then for communication a new ID is referenced by the function.
listen_id	Specifies the corresponding listening request identifier for the `RDMA_CM_EVENT_CONNECT_REQUEST` event type.
event	Specifies the type of communication event that occurred.

Item	Description
status	Returns asynchronous error information associated with an event. The status is zero if the operation was successful, otherwise the status value is non-zero and is either set to an **errno** or a transport specific value. For details on transport specific status values, see the event type information below.
param	Provides additional details based on the type of event. You must select the *conn* subfield based on the **rdma_port_space** function of the `rdma_cm_id` identifier that is associated with the event.

Connection Event Data

The event parameters are related to the connected queue pair (QP) services and the `RDMA_PS_TCP` event type. The connection related event data is valid for `RDMA_CM_EVENT_CONNECT_REQUEST` and `RDMA_CM_EVENT_ESTABLISHED` event types.

Item	Description
private_data	References any user-specified data that is associated with the event. The data referenced by this field matches the value specified by the remote side when running the **rdma_connect** or **rdma_accept** functions. If the event does not include any private data, the *private_data* field is NULL. The buffer referenced by this pointer is deallocated when running the **rdma_ack_cm_event** function.
private_data_len	Specifies the size of the private data buffer. **Note:** The size of the private data buffer might be larger than the amount of private data sent by the remote side. Any additional space in the buffer is zeroed out.
responder_resources	Specifies the number of responder resources that is requested by the recipient. The *responder_resources* field must match the initiator depth specified by the remote node when running the **rdma_connect** and **rdma_accept** functions.
initiator_dept	Specifies the maximum number of outstanding RDMA read operations that the recipient holds. The *initiator_dept* field must match the responder resources specified by the remote node when running the **rdma_connect** and **rdma_accept** functions.
flow_control	Indicates whether the hardware level flow control is provided by the sender. This value is specific to the InfiniBand architecture.
retry_count	Indicates the number of times that the recipient must retry a send operation specific to the `RDMA_CM_EVENT_CONNECT_REQUEST` events. This value is specific to the InfiniBand architecture.
rnr_retry_count	Indicates the number of times that the recipient must retry receiver not ready (RNR) NACK errors. This value is specific to the InfiniBand architecture.
srq	Specifies whether the sender is using a shared-receive queue. Currently, the field is not supported.
qp_num	Indicates the remote QP number for the connection.

Event Types

The following types of communication events are reported.

Item	Description
`RDMA_CM_EVENT_ADDR_RESOLVED`	Indicates that the address resolution (`rdma_resolve_addr`) completed successfully.
`RDMA_CM_EVENT_ADDR_ERROR`	Indicates that the address resolution (`rdma_resolve_addr`) failed.

Item	Description
RDMA_CM_EVENT_ROUTE_RESOLVED	Indicates that the route resolution (rdma_resolve_route) completed successfully.
RDMA_CM_EVENT_ROUTE_ERROR	Indicates that the route resolution (rdma_resolve_route) failed.
RDMA_CM_EVENT_CONNECT_REQUEST	Indicates that there is a new connection request on the passive side.
RDMA_CM_EVENT_CONNECT_RESPONSE	Indicates that the there is a successful response to a connection request on the active side. It is generated on rdma_cm_id identifiers without a QP associated with them.
RDMA_CM_EVENT_CONNECT_ERROR	Indicates that an error has occurred trying to establish a connection. this event type can be generated on the active or passive side of a connection.
RDMA_CM_EVENT_UNREACHABLE	Indicates that the remote server is not reachable or unable to respond to a connection request on the active side. This option is generated on the active side to notify the user that the remote server is not reachable or unable to respond to a connection request. If this event is generated in response to a UD QP resolution request over InfiniBand, the event status field contains an **errno**, if negative, or the status result carried in the IB CM SIDR REP message.
RDMA_CM_EVENT_REJECTED	Indicates that a connection request or response was rejected by the remote end point. The event status field contains the transport specific reject reason if available. For InfiniBand, this event carries the reason for rejection in the IB CM REJ message.
RDMA_CM_EVENT_ESTABLISHED	Indicates that a connection is established with the remote end point.
RDMA_CM_EVENT_DISCONNECTED	Indicates that the connection is disconnected.
RDMA_CM_EVENT_DEVICE_REMOVAL	Indicates that the local RDMA device associated with the rdma_cm_id identifier is removed. When you receive this event, you must destroy the associatedrdma_cm_id identifier.
RDMA_CM_EVENT_TIMEWAIT_EXIT	Indicates that the QP associated with a connection has exited its timewait state and is ready to be reused. After a QP is disconnected, it is maintained in a timewait state to allow any in flight packets to exit the network. After the timewait state is complete, the rdma_cm identifier reports this event.

rdma_ack_cm_event

Frees a communication event.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_ack_cm_event(struct rdma_cm_event *event);
```

Description

All events that are allocated by the **rdma_get_cm_event** function must be released. There must be a one-to-one correspondence between the events that are retrieved from a queue and events being released. The **rdma_ack_cm_event** function releases the event structure and any memory that it references.

Parameters

Item	Description
event	Specifies the event to be released.

Return Values

The **rdma_ack_cm_event** function returns the following values:

Item	Description
0	On success.
-1	If an error occurs, **errno** specifies the reason for failure.

rdma_event_str

Returns a string representation of an RDMA CM event.

Syntax

```
#include <rdma/rdma_cma.h>
const char *rdma_event_str(enum rdma_cm_event_type event);
```

Description

The **rdma_event_str** function returns a string representation of an asynchronous event.

Parameters

Item	Description
event	Specifies an asynchronous event.

Return Values

The **rdma_event_str** function returns a pointer to a static character string that corresponds to the event.

Data transfer operations

Lists the verbs that are used in data transfer operations such to retrieve the work request, send and receive work request, post the status of the request.

rdma_get_recv_comp

Retrieves a completed work request for the receive operation.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_get_recv_comp (struct rdma_cm_id *id, struct ibv_wc *wc);
```

Description

The **rdma_get_recv_comp** operation specifies the information about the completed request. The operation returns the information by using the *wc* parameter, and uses the `wr_id` identifier to set the context of the request.

Notes: The **rdma_get_recv_comp** operation polls the receive completion queue that is associated with an **rdma_cm_id** identifier. If the queue is not complete, the call is blocked until the request is completed.

This call must be used on the **rdma_cm_id** identifiers that do not share change queues (CQs) with other **rdma_cm_id** identifiers, and must maintain separate CQs to send and receive completed work request.

Parameters

Item	Description
id	Specifies a reference to a communication identifier to check the completion of the request.
wc	Specifies a reference to a work completion structure that must be filled.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

rdma_get_request

Retrieves the connection request event that is pending.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_get_request (struct rdma_cm_id *listen, struct rdma_cm_id **id);
```

Description

Retrieves a connection request event that is in pending state. If no requests are in a pending state, the call is blocked until an event is received.

Notes: The `rdma_get_request` call must be used on the listening `rdma_cm_id` communication identifiers that are operating synchronously. You receive a new `rdma_cm_id` identifier that represents the connection request on successful completion of the call. The new `rdma_cm_id` identifier is related to event information that is associated with the request until one of the following requisites is satisfied:

- The `rdma_reject` and `rdma_accept` operations are called.
- The `rdma_destroy_id` identifier is called on the newly created identifier.

If queue pair (QP) attributes are associated with the listening endpoint, the `rdma_cm_id` identifier that is returned is related an allocated to queue pair (QP).

Parameters

Item	Description
id	Specifies the communication identifier that is associated to the new connection.
listen	Specifies the communication identifier that is listening.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_get_send_comp

Retrieves a completed request for send, read, or write operation.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_get_send_comp (struct rdma_cm_id *id, struct ibv_wc *wc);
```

Description

Retrieves a completed work request for a send, RDMA read, or RDMA write operation. Information about the completed request is returned by using the *wc* parameter, which has the `wr_id` identifier set to the context of the request.

Notes: The `rdma_get_send_comp` operation polls the send completion queue that is associated with an `rdma_cm_id` identifier. If a completion request is not found, the `rdma_get_send_comp` call blocks the queue until a request is completed. The `rdma_get_send_comp` call must be used on `rdma_cm_id` identifiers that do not share change queues (CQs) with other `rdma_cm_id` identifiers, and the function maintains separate CQs for send and receive completion requests.

Parameters

Item	Description
id	Specifies a reference to a communication identifier to check for completions.
wc	Specifies a reference to a work completion structure that must be filled.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

rdma_post_read

Posts a work request for RDMA read operation.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_read (struct rdma_cm_id *id, void *context, void *addr, size_t length,struct
ibv_mr *mr,
int flags, uint64_t remote_addr, uint32_t rkey);
```

Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the remote memory region are read into the local data buffer.

Notes: The remote and local data buffers must be registered before running the read operation, and the buffers must be registered until the read operation is completed. The read operation does not post the work request to an `rdma_cm_id` identifier or to the corresponding queue pair until it is connected. The user-defined context that is associated with the read request is returned by using the work completion `wr_id` identifier and the work request identifier field.

Parameters

Item	Description
addr	Specifies the address of the local destination of the read request.
context	Specifies the user-defined context that is associated with the request.
flags	Specifies the optional flags that are used to control the read operation.
id	Specifies a reference to a communication identifier where the request is posted.
rkey	Specifies the registered memory key that is associated with the remote address.
length	Specifies the length of the read operation.
mr	Specifies the registered memory region that is associated with the local buffer.
remote_addr	Specifies the address of the remote registered memory to read the address.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_post_readv

Posts a work request to the send queue for RDMA read operation.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_readv (struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge, int
flags,
uint64_t remote_addr, uint32_t rkey);
```

Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the remote memory region are read into the local data buffer.

Notes: The remote and local data buffers must be registered before running the read operation and the buffers must be registered until the read operation is completed. The read operation does not post the work request to an`rdma_cm_id` identifier or the corresponding queue pair until the connection is established. The user-defined context that is associated with the read request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

Parameters

Item	Description
context	Specifies the user-defined context that is associated with the request.
flags	Specifies the optional flags that is used to control the read operation.

Item	Description
id	Specifies a reference to a communication identifier where the request is posted.
nsge	Specifies the number of scatter-gather array entries that are present.
rkey	Specifies the registered memory key that is associated with the remote address.
remote_addr	Specifies the address of the remote registered memory to read the address.
sgl	Specifies a scatter-gather list of the destination buffers that is associated with the read operation.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

rdma_post_recv

Posts a work request to receive an incoming message.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_recv (struct rdma_cm_id *id, void *context, void *addr, size_t length,
struct ibv_mr *mr);
```

Description

Posts a work request to the receive queue of the queue pair that is associated with the `rdma_cm_id` identifier. The posted buffer is queued to receive an incoming message that is sent by the remote peer.

Notes: You must make sure that a receive buffer is posted. The receive buffer must be large enough to contain all the sent data before the peer posts the corresponding send message. You must register the message buffer before it is posted by using the `mr` parameter specifying the registration. The buffer must be registered until the receive operation is completed. The messages can be posted to an `rdma_cm_id` identifier after a queue pair is associated with the message. If the `rdma_cm_id` identifier is allocated by using the `rdma_create_id` identifier, a queue pair is bound to an `rdma_cm_id` identifier after calling the`rdma_create_ep` operation or `rdma_create_qp` operation. The user-defined context that is associated with the receive request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

Parameters

Item	Description
addr	Specifies the address of the memory buffer to post the work request.
context	Specifies the user-defined context that is associated with the request.
id	Specifies a reference to a communication identifier where the message buffer is posted.
length	Specifies the length of the memory buffer.
mr	Specifies the registered memory region that is associated with the posted buffer.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

rdma_post_recvv

Posts a work request to the send queue for RDMA read operation.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_recvv (struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge);
```

Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the remote memory region is read into the local data buffer.

Notes: You must make sure that a receive buffer is posted. The receive buffer must be large enough to contain all the sent data before the peer posts the corresponding send message. You must register the message buffer before it is posted by using the `mr` parameter by specifying the registration. The buffer must be registered until the receive operation is completed. The messages can be posted to an `rdma_cm_id` identifier after a queue pair is associated with the message. A queue pair is bound to an `rdma_cm_id` identifier after calling the`rdma_create_ep` operation or `rdma_create_qp` operation, if the `rdma_cm_id` identifier is allocated by using the `rdma_create_id` identifier. The user-defined context that is associated with the receive request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

Parameters

Item	Description
context	Specifies the user-defined context that is associated with the request.
id	Specifies a reference to a communication identifier where the request is posted.
nsge	Specifies the number of scatter-gather array entries that are present.
sgl	Specifies a scatter-gather list of the destination buffers that is associated with the read operation.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_post_send

Posts a work request to send a message.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_send (struct rdma_cm_id *id, void *context, void *addr, size_t length, struct
ibv_mr *mr,
int flags);
```

Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the posted buffer are sent to the remote peer of a connection.

Notes: You must make sure that the remote peer posts a receive request before processing the send operations. If the send request is using inline data, the message buffer must be registered before being posted with the `mr` parameter by specifying the registration. The buffer must remain registered until the send operation is completed. The send operation cannot be posted to an `rdma_cm_id` identifier or the corresponding queue pair until the send operation is connected. The user-defined context that is associated with the send request is returned to the user by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

Parameters

Item	Description
addr	Specifies the address of the memory buffer to post the work request.
context	Specifies the user-defined context that is associated with the request.
flags	Specifies the optional flags that are used to control the send operation.
id	Specifies a reference to a communication identifier where the message buffer is posted.
length	Specifies the length of the memory buffer.
mr	Specifies the optional registered memory region that is associated with the posted buffer.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_post_sendv

Posts a work request to send a message.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_sendv (struct rdma_cm_id *id, void *context, struct ibv_sge *slg, int nsge,
int flags);
```

Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identified. The contents of the posted buffer are sent to the remote peer of a connection.

Notes: You must make sure that the remote peer posts a receive request before processing the send operations. If the send request is using inline data, the message buffer must be registered before being posted with the`mr` parameter by specifying the registration. The buffer must remain registered until the send operation is completed. The send operation cannot be posted to an `rdma_cm_id` identifier or the corresponding queue pair until the send operation is connected. The user-defined context that is associated with the send request is returned to the user by using the work completion`wr_id` identifier, the work request identifier, and the work request identifier field.

Parameters

Item	Description
context	Specifies the user-defined context that is associated with the request.
flags	Specifies the optional flags that are used to control the send operation.
id	Specifies a reference to a communication identifier where the message buffer is posted.
nsge	Specifies the number of scatter-gather entries in the *slg* array.
slg	Specifies a scatter-gather list of memory buffers that is posted as a single request.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_post_ud_send

Posts a work request to send a datagram.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_ud_send (struct rdma_cm_id *id, void *context, void *addr, size_t length,
struct ibv_mr *mr, int flags, struct ibv_ah *ah, uint32_t remote_qpn);
```

Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identified. The contents of the posted buffer is sent to the specified destination of the queue pair.

Notes: You must make sure that the remote peer posts a receive request before processing the send operations. If the send request is using inline data, the message buffer must be registered before being posted with the `mr` parameter by specifying the registration. The buffer must remain registered until the send operation is completed. The send operation cannot be posted to an `rdma_cm_id` identifier or the corresponding queue pair until the send operation is connected. The user-defined context that is associated with the send request is returned to the user by using the work completion`wr_id` identifier, work request identifier, and field.

Parameters

Item	Description
addr	Specifies the address of the memory buffer to post the work request.
ah	Specifies an address handle that describes the address of the remote node.
context	Specifies the user-defined context that is associated with the request.
flags	Specifies the optional flags that are used to control the send operation.
id	Specifies a reference to a communication identifier where the message buffer is posted.

Item	Description
length	Specifies the length of the memory buffer.
mr	Specifies the optional registered memory region that is associated with the posted buffer.
remote_qpn	Specifies the number of the destination queue pair.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, then **errno** is set to indicate the reason for failure.

rdma_post_write

Posts a work request for RDMA write operation.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_write (struct rdma_cm_id *id, void *context, void *addr, size_t length, struct
ibv_mr *mr,
int flags, uint64_t remote_addr, uint32_t rkey);
```

Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the local data buffer are written into the remote memory region.

Notes: The remote and local data buffers must be registered before you run the write operation. The buffers must be registered until the write operation is complete. The write operation does not post the work request to an `rdma_cm_id` identifier or the corresponding queue pair until it is connected. The user-defined context that is associated with the write request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

Parameters

Item	Description
addr	Specifies the local address of the source that is related to the write request.
context	Specifies the user-defined context that is associated with the request.
flags	Specifies the optional flags that are used to control the write operation.
id	Specifies a reference to a communication identifier where the request is posted.
rkey	Specifies the registered memory key that is associated with the remote address.
length	Specifies the length of the write operation.
mr	Specifies the optional memory region that is associated with the local buffer.
remote_addr	Specifies the address of the remote registered memory to write the data.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_post_writev

Posts a work request for RDMA write operation.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_post_writev (struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge,
int flags, uint64_t remote_addr, uint32_t rkey);
```

Description

Posts a work request to the send queue of the queue pair that is associated with the `rdma_cm_id` identifier. The contents of the local data buffer are written into the remote memory region.

Notes: The remote and local data buffers must be registered before you run the write operation. The remote and local data buffers must be registered until the write operation is completed. The write operation does not post the work request to an `rdma_cm_id` identifier or the corresponding queue pair until it is connected. The user-defined context that is associated with the write request is returned by using the work completion `wr_id` identifier, the work request identifier, and the work request identifier field.

Parameters

Item	Description
context	Specifies the user-defined context that is associated with the request.
flags	Specifies the optional flags that are used to control the write operation.
id	Specifies a reference to a communication identifier where the request is posted.
nsge	Specifies the number of scatter-gather array entries.
remote_addr	Specifies the address of the remote registered memory to write the data.
rkey	Specifies the registered memory key that is associated with the remote address.
sgl	Specifies a scatter-gather list of the source buffer that is related to the write operation.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

Queue Pair Management

Lists the functions that help to manage queue pair (QP) such as to create QP, and to destroy QP.

rdma_create_qp

Allocates a queue pair (QP).

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_create_qp(struct rdma_cm_id *id, struct ibv_pd *pd, struct ibv_qp_init_attr
*qp_init_attr);
```

Description

The **rdma_create_qp** function allocates a queue pair (QP) that is associated with a specified `rdma_cm_id` identifier, and transitions it for sending and receiving.

Notes:

- The `rdma_cm_id` identifier must be associated with a local RDMA device before running the **rdma_create_qp** function, and the protection domain must be for the same device.
- QPs that are allocated to an `rdma_cm_id` identifier are automatically transitioned by the `librdmacm` library through their states. The QP is ready to handle posting of received data after the QP is allocated. If the QP is not connected, it is ready to post send data.
- If a protection domain is not specified then the - pd parameter is NULL, then the `rdma_cm_id` identifer is created by using a default protection domain. One default protection domain is allocated per RDMA device. The initial QP attributes are specified by using the`qp_init_attr` parameter. The `send_cq` and `recv_cq` fields in the `ibv_qp_init_attr` are optional. If a send or receive completion queue (CQ) is not specified, then a CQ is allocated by the `rdma_cm` for the QP, along with corresponding completion channels. Completion channels and CQ data created by the `rdma_cm` can be accessed by user by using the `rdma_cm_id` structure. The actual capabilities and properties of the QP that is created is returned to the user through the `qp_init_attr` parameter.

Parameters

Item	Description
id	Specifies the communication identifier to create.
pd	Specifies the optional protection domain for the QP.
qp_init_attr	Specifies the initial QP attributes.

Return Values

The **rdma_destroy_event_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

rdma_destroy_qp

Releases a queue pair (QP).

Syntax

```
#include <rdma/rdma_cma.h>
void rdma_destroy_qp(struct rdma_cm_id *id);
```

Description

The **rdma_destroy_qp** function destroys a QP that is allocated to an `rdma_cm_id` identifier.

Note: You must destroy any QP that is associated with an `rdma_cm_id` identifier before deleting the ID.

Parameters

Item	Description
id	Specifies the RDMA identifier.

Return Value

The **rdma_destroy_event_channel** function returns 0 on success, or -1 on error. If an error occurs, **errno** indicates the reason for failure.

Device Management

Lists the functions that is used to manage devices, which includes to get devices, and free devices.

rdma_get_devices

Gets a list of RDMA devices that are available.

Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_context **rdma_get_devices(int *num_devices);
```

Description

The **rdma_get_devices** function returns a NULL-terminated array of open RDMA devices. You can use this routine to allocate resources on specific RDMA devices that is shared with multiple `rdma_cm_id` identifiers.

Note: The returned array must be released by running the **rdma_free_devices** function. Devices remain opened when the **librdmacm** library is loaded.

Parameters

Item	Description
num_devices	Specifies the number of devices that are returned if the value is not NULL.

Return Values

The **rdma_get_devices** function returns an array of available RDMA devices, or NULL if the request fails. If an error occurs, **errno** indicates the reason for failure.

rdma_free_devices

Frees the list of devices that are returned by the **rdma_get_devices** function.

Syntax

```
#include <rdma/rdma_cma.h>
void rdma_free_devices(struct ibv_context **list);
```

Description

The **rdma_free_devices** function frees the device array that is returned by the **rdma_get_devices** function.

Parameters

Item	Description
list	Specifies the list of devices that are returned from the **rdma_get_devices** function.

Return Value

There is no return value.

Memory region management

Lists the functions that is used to manage memory, which includes to register memory, to provide read and write access to memory, and to register the buffer for sending and receiving messages.

rdma_dereg_mr

Deregisters a memory region that is registered.

Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_mr * rdma_dereg_mr (struct ibv_mr *mr);
```

Description

Deregisters a memory buffer that is registered for RDMA or message operations. You must run the `rdma_dereg_mr` call for all registered memory that is associated with an `rdma_cm_id` identifier before you destroy the `rdma_cm_id` identifier.

Note: All memory buffers that is registered with an `rdma_cm_id` identifier are associated with the protection domain that is associated with the ID. You must deregister all registered memory before the protection domain can be destroyed.

Parameters

Item	Description
mr	Specifies a reference to a registered memory buffer.

Return Values

Returns 0 on success, or -1 on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_reg_msgs

Registers the data buffer for sending or receiving messages.

Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_mr * rdma_reg_msgs (struct rdma_cm_id *id, void *addr, size_t length);
```

Description

Registers an array of memory buffers that are used for sending and receiving messages or for RDMA operations. Memory buffers that are registered by using the `rdma_reg_msgs` function can be posted to an `rdma_cm_id` identifier by using one of the following operations:

- Run the **rdma_post_send** operation
- Run the **rdma_post_recv** operation
- Specify the buffer as the target of an RDMA read operation
- Specify the buffer as the source of an RDMA write request

Note: The `rdma_reg_msgs` operation registers an array of data buffers that are used to send and receive messages on a queue pair that is associated with an `rdma_cm_id` identifier. The memory buffer is registered with the protection domain that is associated with the identifier. The start of the data buffer array is specified by using the `addr` parameter, and the total size of the array is specified by the `length` parameter. All data buffers must be registered before being posted as a work request. You must unregister all the registered memory by using the `rdma_dereg_mr` operation.

Parameters

Item	Description
addr	Specifies the address of the memory buffer to register.
id	Specifies a reference to a communication identifier where the message buffer must be used.
length	Specifies the total length of the memory to register.

Return Values

Returns a reference to the registered memory region on success, or NULL on error. If an error occurs, **errno** is set to indicate the reason for failure.

rdma_reg_read

Registers the data buffer for remote direct memory access (RDMA) read access.

Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_mr * rdma_reg_read (struct rdma_cm_id *id, void *addr, size_t length);
```

Description

Registers a memory buffer that is accessed by a remote direct memory access (RDMA) read operation. Memory buffers that are registered by using the **rdma_reg_read** operation can be targeted in an RDMA read request. The memory buffer is specified on the remote side of an RDMA connection as the `remote_addr` parameter of **rdma_post_read** operation, or a similar operation.

Notes: The **rdma_reg_read** operation registers a data buffer that is the target of an RDMA read operation on a queue pair that is associated with an **rdma_cm_id** identifier. The memory buffer is registered with the protection domain that is associated with the identifier. The start of the data buffer array is specified by using the `addr` parameter, and the total size of the array is specified by the `length` parameter. All data buffers must be registered before being posted as a work request. You must unregister all the registered memory by using the **rdma_dereg_mr** operation.

Parameters

Item	Description
addr	Specifies the address of the memory buffer to register.
id	Specifies a reference to a communication identifier where the message buffer must be used.
length	Specifies the total length of the memory to register.

Return Values

Returns a reference to the registered memory region on success, or NULL on error. If an error occurs, the **errno** is set to indicate the reason for failure.

rdma_reg_write

Registers the data buffer for remote direct memory access (RDMA) write access.

Purpose

Syntax

```
#include <rdma/rdma_cma.h>
truct ibv_mr * rdma_reg_write (struct rdma_cm_id *id, void *addr, size_t length);
```

Description

Registers a memory buffer that is accessed by a remote RDMA write operation. Memory buffers that are registered by using the **rdma_reg_write** operation can be targeted in an RDMA write request. The memory buffer is specified on the remote side of an RDMA connection as the `remote_addr` parameter of **rdma_post_write** operation, or a similar operation.

Notes: The **rdma_reg_write** operation registers a data buffer that is the target of an RDMA write operation on a queue pair that is associated with an **rdma_cm_id** identifier. The memory buffer is registered with the protection domain that is associated with the identifier. The start of the data buffer array is specified by using the `addr` parameter, and the total size of the array is specified by the `length` parameter. All data buffers must be registered before being posted as a work request. You must unregister all the registered memory by using the **rdma_dereg_mr** operation.

Parameters

Item	Description
addr	Specifies the address of the memory buffer to register.
id	Specifies a reference to a communication identifier where the message buffer must be used.
length	Specifies the total length of the memory to register.

Return Values

Returns a reference to the registered memory region on success, or NULL on error. If an error occurs, the **errno** is set to indicate the reason for failure.

Libibverbs Library

The libibverbs library enables user-space processes to use remote direct memory access (RDMA) verbs as described in the InfiniBand Architecture Specification.

You can find information about the `libibverbs` library in the `/usr/include/rdma/verbs.h` file that is delivered with the `libibverbs` library sources.

Man pages are created to describe the various interfaces and test programs. For a full list, you can refer to the **verbs** man page.

Returned error rules

Lists the errors returned by the Libibverbs library.

The values returned by the commands and their interpretation are as follows:

- The commands return 0 on success.
- The commands return NULL, -1, or the value **errno** that indicates the reason of failure.
- The commands return **ENOSYS** when the verb is not supported.

Supported Verbs

Lists the supported verbs and their functions for the `Libibverbs` library.

Device management

Lists the functions that manage devices for the `libibverbs` library.

ibv_get_device_list, ibv_free_device_list

Obtains and releases the list of available RDMA devices.

Syntax

```
#include <rdma/verbs.h>
struct ibv_device **ibv_get_device_list(int *num_devices);
void ibv_free_device_list(struct ibv_device **list);
```

Description

The **ibv_get_device_list()** function returns a NULL-terminated array of RDMA devices that are available. The argument *num_devices* is optional and if it is NULL, it is set to the number of devices returned in the array.

The **ibv_free_device_list()** function frees the array of devices list that is returned by the **ibv_get_device_list()** function.

Note: The client code must open all the devices it intends to use with the **ibv_open_device()** operation before running the **ibv_free_device_list()** function. When the **ibv_free_device_list()** function frees the array, the system can use the open devices, and the pointers to unopened devices is no longer valid.

Errors

Error	Descriptor
EPERM	Permission denied.
ENOSYS	No kernel support for RDMA.
ENOMEM	Insufficient memory to complete the operation.

Output Parameters

Item	Description
num_devices	(Optional) If not null, the number of devices returned in the array is stored in this parameter.

Return Value

The **ibv_get_device_list()** function returns the array of available RDMA devices, or NULL if the request fails. If no devices are found then **num_devices** is set to 0, and a non-NULL is returned.

The **ibv_free_device_list()** function returns no value.

ibv_get_device_name

Obtains the name of the RDMA device.

Syntax

```
#include <rdma/verbs.h>
const char *ibv_get_device_name(struct ibv_device *device);
```

Description

The **ibv_get_device_name** function returns a pointer to the device name that is contained within the struct **ibv_device**.

Parameters

Item	Description
device	Specifies the struct **ibv_device** for the required device.

Return Value

The **ibv_get_device_list()** function returns a pointer to the device name on success, and NULL if the request fails.

ibv_get_device_guid

Returns the string that describes the **event_type**, **node_type**, and **port_state** for the **enum** values.

Syntax

```
#include <rdma/verbs.h>
uint64_t ibv_get_device_guid(struct ibv_device *device);
```

Description

The **ibv_get_device_guid** function returns a 64-bit global unique identifier (GUID) for the devices in the network byte order.

Parameters

Item	Description
device	Specifies the struct **ibv_device** for the device.

Return Value

The **ibv_get_device_guid** function returns **uint64_t** on success, and **0** on failure.

If the device is NULL, the open or write operation failed on the/dev/rdma/ofed_adm administrator device.

ibv_open_device, ibv_close_device

Opens and closes an remote device memory access (RDMA) device context.

Syntax

```
#include <rdma/verbs.h>
struct ibv_context *ibv_open_device(struct ibv_device *device);
int ibv_close_device(struct ibv_context *context);
```

Description

The **ibv_open_device()** function opens the device*device*, and creates a context for further use.

The **ibv_close_device()** function closes the device context*context*.

Note: The **ibv_close_device()** function does not release all the resources that are allocated by using the parameter *context*. To avoid resource leaks, you must release all the associated resources before closing a context.

Parameter

Item	Description
devices	Specifies the struct **ibv_device** for the required device.

Return Value

The **ibv_open_device**and **ibv_close_device** functions return a verb context that can be used for future operations on the device on successful completion. The function returns NULL if the device is NULL, or if the open operation fails.

ibv_query_device

Queries the attributes of an RDMA device.

Syntax

```
#include <rdma/verbs.h>
int ibv_query_device(struct ibv_context *context, struct ibv_device_attr *device_attr)
```

Description

The **ibv_query_device()** function returns the attributes of the device with context *context*. The parameter *device_attr* is a pointer to an **ibv_device_attr** struct as defined in the <rdma/verbs.h> file.

Note: The maximum values that are returned by the **ibv_query_device()** function are the upper limits of the supported resources by the device. It is not possible to use these maximum values because the actual number of any resource that can be created is limited by the system configuration, the amount of host memory, user permissions, and the amount of resources in use.

Input Parameter

Item	Description
context	Specifies the struct `ibv_context` from the **ibv_open_device** function.

Output Parameter

Item	Description
device_attr	Specifies the struct **ibv_device_attr** that contains the device attributes.

Return Values

Item	Description
0	On success.
errno	On failure.

ibv_query_port

Queries the attributes of an RDMA port.

Syntax

```
#include <rdma/verbs.h>
int ibv_query_port(struct ibv_context *context, uint8_t port_num, struct ibv_port_attr
*port_attr)
```

Description

The **ibv_query_port()** function returns the attributes of port *port_num* for device context *context* through the pointer *port_attr*. The parameter *port_attr* is an **ibv_port_attr** struct, as defined in the <rdma/verbs.h> file.

The struct **ibv_port_attr** is as follows:

```
struct ibv_port_attr {
enum ibv_port_state     state;          /* Logical port state */
enum ibv_mtu            max_mtu;        /* Max MTU supported by port */
enum ibv_mtu            active_mtu;     /* Actual MTU */
int                     gid_tbl_len;    /* Length of source GID table */
uint32_t                port_cap_flags; /* Port capabilities */
uint32_t                max_msg_sz;     /* Maximum message size */
uint32_t                bad_pkey_cntr;  /* Bad P_Key counter */
uint32_t                qkey_viol_cntr; /* Q_Key violation counter */
uint16_t                pkey_tbl_len;   /* Length of partition table */
uint16_t                lid;            /* Base port LID */
uint16_t                sm_lid;         /* SM LID */
uint8_t                 lmc;            /* LMC of LID */
uint8_t                 max_vl_num;     /* Maximum number of VLs */
uint8_t                 sm_sl;          /* SM service level */
uint8_t                 subnet_timeout; /* Subnet propagation delay */
uint8_t                 init_type_reply;/* Type of initialization performed by SM */
uint8_t                 active_width;   /* Currently active link width */
uint8_t                 active_speed;   /* Currently active link speed */
uint8_t                 phys_state;     /* Physical port state */
uint8_t                 link_layer;     /* link layer protocol of the port */
};
```

Input Parameters

Item	Description
context	Specifies the struct **ibv_context** from the **ibv_open_device** function.

Item	Description
port_num	Specifies the physical port number (1 is the first port).

Output Parameter

Item	Description
port_attr	Specifies the struct **ibv_port_attr** that contains the port attributes.

Return Values

Item	Description
0	On success.
errno	On failure.

ibv_query_pkey

Queries the P_Key table of an remote direct memory access (RDMA) port.

Syntax

```
#include <rdma/verbs.h>
int ibv_query_pkey(struct ibv_context *context, uint8_t port_num, int index, uint16_t *pkey)
```

Description

The **ibv_query_pkey()** function returns the P_Key value in the entry *index* of port *port_num* for device context *context* through the pointer *pkey*.

Input Parameters

Item	Description
context	Specifies the valid context pointer that is returned by the **ibv_open_device()** function.
port_num	Specifies the valid port number for the device that is returned by the **ibv_query_device()** function.
index	Specifies the valid index for the *port_num* parameter from attributes that are returned by the **ibv_query_port()** function.

Output Parameter

Item	Description
pkey	Specifies the valid pointer to store protection key.

Return Values

Item	Description
0	On success.
-1	The request fails because of the following reasons: • The *context* or *pkey* parameter is NULL • The open or write operation failed on the `/dev/rdma/ofed_adm` administrator device.

ibv_query_gid

Gets the group ID (GID) that is the network interface controller (NIC)'s Media Control Access (MAC) address.

Syntax

```
#include <rdma/verbs.h>
int ibv_query_gid(struct ibv_context *context, uint8_t port_num, int index, union ibv_gid *gid)
```

Description

The **ibv_query_gid()** function returns the MAC address of the NIC in the `subnet_prefix` parameter and 0 in the `interface_id` identifier.

Input Parameters

Item	Description
context	Specifies the context pointer that is returned by the **ibv_open_device()** function.
port_num	Specifies the port number for the device that is returned by the **ibv_query_device()** function.
index	Specifies the index for the *port_num* parameter that is derived from the attributes that are returned by the **ibv_query_port()** function.

Output Parameter

Item	Description
gid	Specifies the pointer where the group ID (GID) can be stored.

Return Values

Item	Description
0	On success.
-1	The request fails because of one of the following reasons: • The *context* or *gid* parameter is NULL. • The open or write operation failed on the `/dev/rdma/ofed_adm` administrator device.

```
ibv_gid
union ibv_gid
{
       uint8_t      raw[16];
       struct
       {
           uint64_t subnet_prefix;
           uint64_t interface_id;
       } global;
};
```

Queue pair management

Lists the functions that are used to manage the queue.

ibv_create_qp, ibv_destroy_qp

Creates or destroys a queue pair (QP).

Syntax

```
#include <rdma/verbs.h>
struct ibv_qp *ibv_create_qp(struct ibv_pd *pd,
struct ibv_qp_init_attr *qp_init_attr);int ibv_destroy_qp(struct ibv_qp *
qp)
```

Description

The **ibv_create_qp()** function creates a queue pair (QP) that is associated with the *pd* protection domain. The *qp_init_attr* argument is an `ibv_qp_init_attr` struct that is defined in the `<rdma/verbs.h>` file.

Name of the Struct	Item	File name	Description
struct ibv_qp_init_attr {			
	void	*qp_context;	/*Associated context of the QP*/
	struct ibv_cq	*send_cq;	/*CQ to be associated with the Send Queue (SQ)*/
	struct ibv_cq	*recv_cq;	/*CQ to be associated with the Receive Queue (RQ)*/
	struct ibv_srq	*srq;	/*Not Supported*/
	struct ibv_qp_cap	cap;	/*QP capabilities*/
	enum ibv_qp_type	qp_type;	/* QP Transport Service Type: IBV_QPT_RC, IBV_QPT_UC, IBV_QPT_UD, IBV_QPT_XRC or IBV_QPT_RAW_PACKET */
	int	sq_sig_all;	/*If set, each Work Request (WR) submitted to the SQ generates a completion entry*/
	struct ibv_xrc_domain	xrc_domain;	/*Not supported*/
struct ibv_qp_cap {			
	uint32_t	max_send_wr;	/*Requested maximum number of outstanding WRs in the SQ*/
	uint32_t	max_recv_wr;	/*Requested maximum number of outstanding WRs in the RQ*/
	uint32_t	max_send_sge;	/*Requested maximum number of Scatter-gather elements in a WR in the SQ*/
	uint32_t	max_recv_sge;	/*Requested maximum number of Scatter-gather elements in a WR in the SQ*/
	uint32_t	max_inline_data;	/*Requested max number of data (bytes)that can be posted inline to the SQ, otherwise 0*/

Input Parameters

Item	Descriptor
pd	struct **ibv_pd** from **ibv_alloc_pd**.
qp_init_attr	Initial attributes of queue pair.

Output Parameters

Item	Description
qp_init_attr	Actual values that are entered.

Return Value

The **ibv_create_qp()** function returns a pointer to the created QP on success, or NULL if the request fails.

The **ibv_destroy_qp()** function returns 0 on success, or **errno** on failure that indicates the reason for failure.

ibv_modify_qp

Modifies the attributes of a queue pair (QP).

Syntax

```
#include <rdma/verbs.h>

int ibv_modify_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, int ibv_qp_attr_mask attr_mask)
```

Queue pairs (QP) must be taken through an incremental sequence of states before using QP for communication.

The following table indicates the QP states:

Item	Descriptor
RESET	Newly created queues that are empty.
INIT	The basic information required for the queue is set and the queue is ready for posting to receive queue.
RTR	The queue is ready to receive. The remote address information is set for the connected QPs, and the QP can receive packets.
RTS	The queue is ready to send. The timeout and retry parameters are set. The QP can send packets.

The state transitions are used with **ibv_modify_qp** function.

Description

The **ibv_modify_qp()** function modifies the attributes of a QP *qp* with the attributes in *attr* parameter according to the *attr_mask* mask . The *attr* parameter is an **ibv_qp_attr** struct, as defined in the<rdma/verbs.h> file.

```
struct ibv_qp_attr {
enum ibv_qp_state       qp_state;               /* Move the QP to this state */
enum ibv_qp_state       cur_qp_state;           /* Assume this is the current QP state */
enum ibv_mtu            path_mtu;               /* Path MTU (valid only for RC/UC QPs) */
enum ibv_mig_state      path_mig_state;         /* Path migration state (valid if HCA supports APM) */
uint32_t                qkey;                   /* Q_Key for the QP (valid only for UD QPs) */
uint32_t                rq_psn;                 /* PSN for receive queue (valid only for RC/UC QPs) */
uint32_t                sq_psn;                 /* PSN for send queue (valid only for RC/UC QPs) */
uint32_t                dest_qp_num;            /* Destination QP number (valid only for RC/UC QPs) */
int                     qp_access_flags;        /* Mask of enabled remote access operations (valid only
                                                   for RC/UC QPs) */
struct ibv_qp_cap       cap;                    /* QP capabilities (valid if HCA supports QP resizing) */
struct ibv_ah_attr      ah_attr;                /* Primary path address vector (valid only
                                                   for RC/UC QPs) */
struct ibv_ah_attr      alt_ah_attr;            /* Alternate path address vector (valid only
                                                   for RC/UC QPs) */
uint16_t                pkey_index;             /* Primary P_Key index */
uint16_t                alt_pkey_index;         /* Alternate P_Key index */
uint8_t                 en_sqd_async_notify;    /* Enable SQD.drained async notification (Valid only
                                                   if qp_state is SQD) */
uint8_t                 sq_draining;            /* Is the QP draining? Irrelevant for ibv_modify_qp() */
uint8_t                 max_rd_atomic;          /* Number of outstanding RDMA reads & atomic operations
                                                   on the destination QP (valid only for RC QPs) */
uint8_t                 max_dest_rd_atomic;     /* Number of responder resources for handling incoming
                                                   RDMA reads & atomic operations (valid only
                                                   for RC QPs)*/
uint8_t                 min_rnr_timer;          /* Minimum RNR NAK timer (valid only for RC QPs) */
```

```
uint8_t                 port_num;               /* Primary port number */
uint8_t                 timeout;                /* Local ack timeout for primary path (valid only
                                                   for RC QPs) */
uint8_t                 retry_cnt;              /* Retry count (valid only for RC QPs) */
uint8_t                 rnr_retry;              /* RNR retry (valid only for RC QPs) */
uint8_t                 alt_port_num;           /* Alternate port number */
uint8_t                 alt_timeout;            /* Local ack timeout for alternate path (valid only
                                                   for RC QPs) */
};
```

The *attr_mask* parameter specifies the QP attributes to be modified. The argument is either 0 or bitwise OR of one or more of the following flags:

IBV_QP_STATE
Modify qp_state

IBV_QP_CUR_STATE
Set cur_qp_state

IBV_QP_EN_SQD_ASYNC_NOTIFY
Set en_sqd_async_notify

IBV_QP_ACCESS_FLAGS
Set qp_access_flags

IBV_QP_PKEY_INDEX
Set pkey_index

IBV_QP_PORT
Set port_num

IBV_QP_QKEY
Set qkey

IBV_QP_AV
Set ah_attr

IBV_QP_PATH_MTU
Set path_mtu

IBV_QP_TIMEOUT
Set timeout

IBV_QP_RETRY_CNT
Set retry_cnt

IBV_QP_RNR_RETRY
Set rnr_retry

IBV_QP_RQ_PSN
Set rq_psn

IBV_QP_MAX_QP_RD_ATOMIC
Set max_rd_atomic

IBV_QP_ALT_PATH
Set the alternative path through alt_ah_attr, alt_pkey_index, alt_port_num, alt_timeout

IBV_QP_MIN_RNR_TIMER
Set min_rnr_timer

IBV_QP_SQ_PSN
Set sq_psn

IBV_QP_MAX_DEST_RD_ATOMIC
Set max_dest_rd_atomic

IBV_QP_PATH_MIG_STATE
Set path_mig_state

IBV_QP_CAP
Set cap

IBV_QP_DEST_QPN
Set dest_qp_num

Notes:

- If any of the modify attributes or the modify mask is invalid, none of the attributes are modified (including the QP state).
- Not all devices support resizing QPs. To determine whether a device supports resizing of the QP, check whether the IBV_DEVICE_RESIZE_MAX_WR bit is set in the device capabilities flags.
- Not all devices support alternate paths. To determine whether a device supports alternate paths, check whether the IBV_DEVICE_AUTO_PATH_MIG bit is set in the device capabilities flags.
- The following tables indicate the type of the QP transport service, the minimum list of attributes that must be changed after changing the QP state from Reset to Init to RTR to RTS state.

```
The types of QP transport service for the IBV_QPT_UD type, follow:

Next state     Required attributes
----------     ---------------------------------------
Init           IBV_QP_STATE, IBV_QP_PKEY_INDEX, IBV_QP_PORT,
               IBV_QP_QKEY
RTR            IBV_QP_STATE
RTS            IBV_QP_STATE, IBV_QP_SQ_PSN

The types of QP transport service for the IBV_QPT_UC type, follow:

Next state     Required attributes
----------     ---------------------------------------
Init           IBV_QP_STATE, IBV_QP_PKEY_INDEX, IBV_QP_PORT,
               IBV_QP_ACCESS_FLAGS
RTR            IBV_QP_STATE, IBV_QP_AV, IBV_QP_PATH_MTU,
               IBV_QP_DEST_QPN, IBV_QP_RQ_PSN
RTS            IBV_QP_STATE, IBV_QP_SQ_PSN

The types of QP transport service for the IBV_QPT_RC type, follow:

Next state     Required attributes
----------     ---------------------------------------
Init           IBV_QP_STATE, IBV_QP_PKEY_INDEX, IBV_QP_PORT,
               IBV_QP_ACCESS_FLAGS
RTR            IBV_QP_STATE, IBV_QP_AV, IBV_QP_PATH_MTU,
               IBV_QP_DEST_QPN, IBV_QP_RQ_PSN,
               IBV_QP_MAX_DEST_RD_ATOMIC, IBV_QP_MIN_RNR_TIMER
RTS            IBV_QP_STATE, IBV_QP_SQ_PSN, IBV_QP_MAX_QP_RD_ATOMIC,
               IBV_QP_RETRY_CNT, IBV_QP_RNR_RETRY, IBV_QP_TIMEOUT

The types of QP transport service for the  IBV_QPT_RAW_PACKET type, follow:

Next state     Required attributes
----------     ---------------------------------------
Init           IBV_QP_STATE, IBV_QP_PORT
RTR            IBV_QP_STATE
RTS            IBV_QP_STATE
```

Indicates the QP transport service types:

Next state	Required attributes
Init	IBV_QP_STATE, IBV_QP_PKEY_INDEX, IBV_QP_PORT, IBV_QP_ACCESS_FLAGS
RTR	IBV_QP_STATE, IBV_QP_AV, IBV_QP_PATH_MTU, IBV_QP_DEST_QPN, IBV_QP_RQ_PSN, IBV_QP_MAX_DEST_RD_ATOMIC, IBV_QP_MIN_RNR_TIMER
RTS	IBV_QP_STATE, IBV_QP_SQ_PSN, IBV_QP_MAX_QP_RD_ATOMIC, IBV_QP_RETRY_CNT, IBV_QP_RNR_RETRY, IBV_QP_TIMEOUT

Input Parameters

Item	Descriptor
qp	Specifies the `ibv_qp` struct for the **ibv_create_qp** function.
attr	Specifies the QP attributes.

Item	Descriptor
attr_mask	Specifies the bit mask. The bit mask defines the attributes within the *attr* parameter that is set for a call.

Return Values

Item	Descriptor
0	On success.
errno	On failure.

ibv_post_recv

Posts a list of work requests (WRs) to a receive queue.

Syntax

```
#include <rdma/verbs.h>
int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr **bad_wr)
```

Description

The **ibv_post_recv()** function posts the linked list of work requests (WRs) starting with the *wr* parameter to the receive queue of the queue pair. The function stops processing WRs from the list at the first failure that can be detected immediately while requests are being posted, and returns the failing WR through the *bad_wr* parameter.

The *wr* argument is an `ibv_recv_wr` struct, as defined in the`<rdma/verbs.h>` file.

```
struct ibv_recv_wr {
        uint64_t                        wr_id;        /* User defined WR ID */
        struct ibv_recv_wr          *next;        /* Pointer to next WR in list, NULL if last
WR */
        struct ibv_sge                  *sg_list;    /* Pointer to the scatter-gather array */
        int                                num_sge;    /* Size of the scatter-gather array
*/
};

struct ibv_sge {
        uint64_t                        addr;        /* Start address of the local memory
buffer */
        uint32_t                        length;      /* Length of the buffer */
        uint32_t                        lkey;      /* Key of the local memory region */
};
```

Note: The buffers that is used by a WR can be safely reused after the request is completed, and a work completion is retrieved from the corresponding completion queue (CQ).

Input Parameters

Item	Descriptor
qp	Specifies the `ibv_qp` struct for the **ibv_create_qp** function.
wr	Specifies the first work request (WR) that contains the receive buffers.

Output Parameter

Return Values

Item	Descriptor
bad_wr	Specifies the pointer to the first rejected WR.

Item	Descriptor
0	On success.
errno	On failure.

ibv_post_send

Posts a list of work requests (WR) to a send queue.

Syntax

```
#include <rdma/verbs.h>
int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr, struct ibv_send_wr **bad_wr)
```

Description

The **ibv_post_send()** function posts the linked list of work requests (WR) starting with the *wr* parameter to the send queue of the queue pair *qp*. The function stops processing the WRs from the list after detecting the first failure while requests are being posted, and returns the failing WR by using the *bad_wr* parameter.

The *wr* argument is an `ibv_send_wr` struct that is defined in the `<rdma/verbs.h>` file.

The transport service types for the operation codes that RC supports, follow:

OPCODE	IBV_QPT_RC
IBV_WR_SEND	Supported
IBV_WR_SEND_WITH_IMM	Supported
IBV_WR_RDMA_WRITE	Supported
IBV_WR_RDMA_WRITE_WITH_IMM	Supported
IBV_WR_RDMA_READ	Supported
IBV_WR_ATOMIC_CMP_AND_SWP	Not supported
IBV_WR_ATOMIC_FETCH_AND_ADD	Not supported

The attribute `send_flags` describes the properties of the WR. It is either 0 or the bitwise OR of one or more of the following flags:

IBV_SEND_FENCE
Sets the fence indicator. The IBV_SEND_FENCE flag is valid only for QPs with the transport service type IBV_QPT_RC.

IBV_SEND_SIGNALED
Sets the completion notification indicator. The IBV_SEND_SIGNALED flag is relevant only if QP is created with the `sq_sig_all` parameter equal to 0.

IBV_SEND_SOLICITED
Sets the solicited event indicator. The IBV_SEND_SOLICITED flag is valid only for send and remote device memory access (RDMA) write functions with immediate effect.

IBV_SEND_INLINE
Sends data in given gather list as inline data in a send WQE. The IBV_SEND_INLINE flag is valid only for send and RDMA write functions. The L_Key parameter is not verified.

Note: The buffers used by a WR can be safely reused after the request is complete. A work completion is retrieved from the corresponding completion queue (CQ).

Input Parameters

Item	Descriptor
qp	Specifies the ibv_qp struct for the **ibv_create_qp** function.
wr	Specifies the first work request (WR).

Output Parameter

Item	Descriptor
bad_wr	Specifies the pointer to the first rejected WR.

Return Values

Item	Descriptor
0	On success.
errno	On failure.

ibv_query_qp

Gets the attributes of a queue pair (QP).

Syntax

```
#include <rdma/verbs.h>
int ibv_query_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr,
                 int attr_mask,
                 struct ibv_qp_init_attr *init_attr);
```

Description

The **ibv_query_qp()** gets the attributes specified in **attr_mask** for the QP and returns them through the pointers attr and init_attr. The argument attr is an **ibv_qp_attr** struct, as defined in <rdma/verbs.h>.

```
struct ibv_qp_attr {
enum ibv_qp_state       qp_state;           /* Current QP state */
enum ibv_qp_state       cur_qp_state;       /* Current QP state - irrelevant for ibv_query_qp */
enum ibv_mtu            path_mtu;           /* Path MTU (valid only for RC/UC QPs) */
enum ibv_mig_state      path_mig_state;     /* Path migration state (valid if HCA supports APM) */
uint32_t                qkey;               /* Q_Key of the QP (valid only for UD QPs) */
uint32_t                rq_psn;             /* PSN for receive queue (valid only for RC/UC QPs) */
uint32_t                sq_psn;             /* PSN for send queue (valid only for RC/UC QPs) */
uint32_t                dest_qp_num;        /* Destination QP number (valid only for RC/UC QPs) */
int                     qp_access_flags;    /* Mask of enabled remote access operations (valid only
                                               for RC/UC QPs) */
struct ibv_qp_cap       cap;                /* QP capabilities */
struct ibv_ah_attr      ah_attr;            /* Primary path address vector (valid only for RC/UC QPs) */
struct ibv_ah_attr      alt_ah_attr;        /* Alternate path address vector (valid only for RC/UC QPs) */
uint16_t                pkey_index;         /* Primary P_Key index */
uint16_t                alt_pkey_index;     /* Alternate P_Key index */
uint8_t                 en_sqd_async_notify; /* Enable SQD.drained async notification - irrelevant for
                                               ibv_query_qp */
uint8_t                 sq_draining;        /* Is the QP draining? (Valid only if qp_state is SQD) */
uint8_t                 max_rd_atomic;      /* Number of outstanding RDMA reads & atomic operations
                                               on the destination QP (valid only for RC QPs) */
uint8_t                 max_dest_rd_atomic; /* Number of responder resources for handling incoming
                                               RDMA reads & atomic operations (valid only for
                                               RC QPs) */
uint8_t                 min_rnr_timer;      /* Minimum RNR NAK timer (valid only for RC QPs) */
uint8_t                 port_num;           /* Primary port number */
uint8_t                 timeout;            /* Local ack timeout for primary path (valid only
                                               for RC QPs) */
```

```
uint8_t                 retry_cnt;          /* Retry count (valid only for RC QPs) */
uint8_t                 rnr_retry;          /* RNR retry (valid only for RC QPs) */
uint8_t                 alt_port_num;       /* Alternate port number */
uint8_t                 alt_timeout;        /* Local ack timeout for alternate path (valid only
                                               for RC QPs) */
};
```

For details on struct **ibv_qp_cap()**, see the description of **ibv_create_qp** function. For details on struct **ibv_ah_attr**, see the description of **ibv_create_ah()** function.

Return Values

On success, the **ibv_query_qp()** function returns 0, or the **errno** on failure that indicates the reason for failure.

ibv_attach_mcast

Attaches and detaches a queue pair (QPs) to or from a multicast group

Syntax

```
#include <rdma/verbs.h>
int ibv_attach_mcast(struct ibv_qp *qp, const union ibv_gid *gid,
                     uint16_t lid);
int ibv_detach_mcast(struct ibv_qp *qp, const union ibv_gid *gid,
                     uint16_t lid);
```

Description

The **ibv_attach_mcast** function attaches the queue pair (QP) to the multicast group that has the MGID gid and MLID lid. The **ibv_detach_mcast** function detaches the QP to the multicast group that has the MGID gid and MLID lid.

Note:

- QPs of Transport Service Type IBV_QPT_UD or IBV_QPT_RAW_PACKET can be attached to multicast groups.
- If a QP is attached to the same multicast group multiple times, the QP receives a single copy of a multicast message.
- To receive multicast messages, a join request for the multicast group must be sent to the subnet administrator (SA). The fabric's multicast routing is configured on receiving the join request to deliver messages to the local port.

Return Values

0
The **ibv_attach_mcast** and **ibv_detach_mcast** functions returns 0 on success.

errno
The **ibv_attach_mcast** and **ibv_detach_mcast** functions returns 0 on failure. **errno** also specifies the reason for failure.

Examples

To use **ibv_attach_mcast** function with RAW ETH QP, use the following program:

```
union ibv_gid mgid;

        memset(&mgid, 0, sizeof(union ibv_gid));
        memcpy(&mgid.raw[10], mmac, 6);
        if (ibv_attach_mcast(qp, &mgid, 0)) {
            printf ("Failed to attach qp to mcast. Errno: %d\n",errno);
            return 1;
         }
```

Completion queue management

Lists the functions that is used to manage the completion queue for the `libibverbs` library.

ibv_create_cq, ibv_destroy_cq

Creates or destroys a completion queue (CQ).

Syntax

```
#include <rdma/verbs.h>
struct ibv_cq *ibv_create_cq(struct ibv_context *context, int cqe, void *cq_context,
struct ibv_comp_channel *channel, int comp_vector)
int ibv_destroy_cq(struct ibv_cq *cq)
```

Description

The **ibv_create_cq()** function creates a completion queue (CQ). A completion queue holds completion queue events (CQE). Each queue pair (QP) has an associated send and receive CQ. A single CQ can be shared for sending, receiving, and sharing across multiple QPs.

The *cqe* parameter defines the minimum size of the queue. The actual size of the queue might be larger than the specified value.

The *cq_context* parameter is a user-defined value. If the value is specified during CQ creation, this value is returned as a parameter in the **ibv_get_cq_event()** function when using a completion channel (CC).

The *channel* parameter is used to specify a CC. A CQ is merely a queue that does not have a built-in notification mechanism. When using a polling paradigm for CQ processing, a CC is not required. Poll the CQ at regular intervals. However, if you want to use a pend paradigm, a CC is required. The CC is a mechanism that allows the user to be notified that a new CQE is on the CQ.

The CQ uses the **comp_vector** parameter for signaling completion events. It must be at least zero and less than the `context->num_comp_vectors` parameter.

The **ibv_destroy_cq()** function destroys the CQ *cq*.

Notes:

- The **ibv_create_cq()** function can create a CQ with a size greater than or equal to the requested size. You can determine the actual size of the function from the *cqe* attribute in the returned CQ.
- The **ibv_destroy_cq()** function fails if any queue pair is still associated with the CQ.

Parameters

Item	Descriptor
context	The **ibv_context** struct for the **ibv_open_device()** function.
cqe	Minimum number of entries that CQ supports.
cq_context	(Optional) Specifies a user-defined value that is returned with completion events.
channel	(Optional) Specifies the completion channel.
comp_vector	(Optional) Specifies the completion vector.

Return Value

The **ibv_create_cq()** function returns a pointer to the CQ, or NULL if the request fails.

The **ibv_destroy_cq()** function returns 0 on success, or the value **errno** on failure, which indicates the reason for failure.

ibv_req_notify_cq

Requests the completion notification on a completion queue (CQ).

Syntax

```
#include <rdma/verbs.h>
int ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only);
```

Description

The **ibv_req_notify_cq()** function requests a completion notification on the completion queue (CQ) *cq* parameter.

When a new CQ entry (CQE) is added to a *cq* parameter, a completion event is added to the completion channel that is associated with the CQ. If the *solicited_only* argument is zero, a completion event is generated for any new CQE. If *solicited_only* parameter is nonzero, an event is generated for a new CQE that is considered solicited. A CQE is solicited if it receives completion for a message that has the solicited event header bit set, or if the status is not successful.

All other successful receive completions or any successful send completion is unsolicited.

Note: The request for a notification is sent once. One completion event is generated for each call that is made to the **ibv_req_notify_cq()** function.

Parameters

Item	Descriptor
cq	Specifies the `ibv_cq` struct for the **ibv_create_cq** function.
solicited_only	Notifies only if the WR is flagged as solicited.

Return Values

Item	Descriptor
0	On success.
errno	On failure.

ibv_poll_cq

Polls a completion queue (CQ).

Syntax

```
#include <rdma/verbs.h>
int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc)
```

Description

The **ibv_poll_cq()** function polls the change queue (CQ) for work completions and returns the first *num_entries* parameter with completions (or all available completions if the CQ contains less than this number) in the *wc* array. The *wc* argument is a pointer to an array of **ibv_wc** struct that is defined in the `<rdma/verbs.h>` file.

```
struct ibv_wc {
uint64_t              wr_id;          /* ID of the completed Work Request (WR) */
enum ibv_wc_status    status;         /* Status of the operation */
enum ibv_wc_opcode    opcode;         /* Operation type specified in the completed WR */
uint32_t              vendor_err;     /* Vendor error syndrome */
uint32_t              byte_len;       /* Number of bytes transferred */
uint32_t              imm_data;       /* Immediate data (in network byte order) */
uint32_t              qp_num;         /* Local QP number of completed WR */
uint32_t              src_qp;         /* Source QP number (remote QP number) of completed
```

```
                                        WR (valid only for UD QPs) */
int                     wc_flags;       /* Flags of the completed WR */
uint16_t                pkey_index;     /* P_Key index (valid only for GSI QPs) */
uint16_t                slid;           /* Source LID */
uint8_t                 sl;             /* Service Level */
uint8_t                 dlid_path_bits; /* DLID path bits (not applicable for multicast
                                           messages) */
};
```

The **wc_flags** attribute describes the properties of the work completion. The flag is either 0 or the bitwise OR of one or more of the following flags:

IBV_WC_GRH
GRH is present.

IBV_WC_WITH_IMM
Immediate data value is valid.

Not all **wc** attributes are always valid. If the completion status is other than IBV_WC_SUCCESS, only the **wr_id**, **status**, **qp_num**, and **vendor_err** attributes are valid.

Note: Each polled completion is removed from the CQ and cannot be returned to it. You must consume work completions at a rate that prevents a CQ overrun from occurrence. In a CQ overrun, the asynchronous **IBV_EVENT_CQ_ERR** event is triggered, and the CQ cannot be used.

Input Parameters

Item	Descriptor
cq	Specifies the `ibv_cq` struct from the **ibv_create_cq** function.
num_entries	Specifies the maximum number of completion queue entries (CQE) to return.

Output Parameters

Item	Descriptor
wc	Specifies the CQE array.

Return Values

On success, the **ibv_poll_cq()** function returns a non-negative value equal to the number of completions found. On failure, a negative value is returned.

ibv_get_cq_event, ibv_ack_cq_events

Gets and acknowledges the completion queue (CQ) events.

Syntax

```
#include <rdma/verbs.h>
int ibv_get_cq_event(struct ibv_comp_channel *channel, struct ibv_cq **cq, void **cq_context);
void ibv_ack_cq_events(struct ibv_cq *cq, unsigned int nevents);
```

Description

The **ibv_get_cq_event()** function waits for the next completion event in the completion event channel. The *cq* argument is used to return the CQ that caused the event and the *cq_context* parameter is used to return the context of the CQ.

The **ibv_ack_cq_events()** function acknowledges the *nevents* events on the CQ *cq* parameter.

Notes:

- All completion events that the **ibv_get_cq_event()** function returns must be acknowledged by using the **ibv_ack_cq_events()** function.
- To avoid competiiton, when you destroy a CQ, the CQ waits for the completion of the events. This action guarantees a one-to-one correspondence between acknowledgements and successful get operation.
- When you call the **ibv_ack_cq_events()** function, it is expensive in the datapath because it must take a mutex. To reduce the cost, a count of the number of events requesting acknowledgement and acknowledging several completion events in one call to the **ibv_ack_cq_events()** function are performed.

Input Parameters

Item	Descriptor
channel	The **ibv_comp_channel** struct for the **ibv_create_comp_channel()** function.

Output Parameters

Item	Descriptor
cq	A pointer to the completion queue (CQ) that is associated with the event.
cq_context	The user-supplied context that is set in the **ibv_create_cq()** function.

Return Value

The **ibv_get_cq_event()** and **ibv_ack_cq_events()** functions return 0 on success, and -1 if the request fails.

Examples

1. The following code example demonstrates one possible way to work with completion events. It performs the following steps:
   a. Preparation:
      i) Creates a CQ.
      ii) Requests notification after creation of a new (first) completion event.
   b. Completion handling routine:
      i) Waits for the completion event and acknowledges it.
      ii) Requests notification for the next completion event.
      iii) Empties the CQ.

   Note: An extra event can be triggered without having a corresponding completion entry in the CQ. This occurs if a completion entry is added to the CQ between requesting for notification and emptying the CQ. Then, the CQ is emptied.

```
cq = ibv_create_cq(ctx, 1, ev_ctx, channel, 0);
if (!cq) {
        fprintf(stderr, "Failed to create CQ\n");
        return 1;
}

/* Request notification before any completion can be created */
if (ibv_req_notify_cq(cq, 0)) {
        fprintf(stderr, "Could not request CQ notification\n");
        return 1;
}
```

```
        .
        .
        .
        /* Wait for the completion event */
        if (ibv_get_cq_event(channel, &ev_cq, &ev_ctx)) {
                fprintf(stderr, "Failed to get cq_event\n");
                return 1;
        }

        /* Ack the event */
        ibv_ack_cq_events(ev_cq, 1);

        /* Request notification upon the next completion event */
        if (ibv_req_notify_cq(cq, 0)) {
                fprintf(stderr, "Could not request CQ notification\n");
                return 1;
        }

        /* Empty the CQ: poll all of the completions from the CQ (if any exist) */
        do {
                ne = ibv_poll_cq(cq, 1, &wc);
                if (ne < 0) {
                        fprintf(stderr, "Failed to poll completions from the CQ\n");
                        return 1;
                }
                if (wc.status != IBV_WC_SUCCESS) {
                        fprintf(stderr, "Completion with status 0x%x was found\n",
wc.status);
                        return 1;
                }
        } while (ne);
```

2. The following code example demonstrates a possible way to work with completion events in nonblocking mode. The code performs the following steps:

   a. Sets the completion event channel in nonblocked mode.

   b. Polls the channel until it has a completion event.

   c. Gets the completion event and acknowledges it.

```
/* change the blocking mode of the completion channel */
flags = fcntl(channel->fd, F_GETFL);
rc = fcntl(channel->fd, F_SETFL, flags | O_NONBLOCK);
if (rc < 0) {
        fprintf(stderr, "Failed to change file descriptor of completion event channel\n");
        return 1;
}
/*
* poll the channel until it has an event and sleep ms_timeout
* milliseconds between any iteration
*/
my_pollfd.fd = channel->fd;
my_pollfd.events = POLLIN;
my_pollfd.revents = 0;

do {

rc = poll(&my_polfd;, 1, ms_timeout);
        } while (rc == 0);
        if (rc &lt; 0){  fprintf(stderr, "poll failed\n");
        return 1;
        }
 ev_cq = cq;
 /* Wait for the completion event */
 if (ibv_get_cq_event(channel, &ev_cq, &ev_ctx)) {
         fprintf(stderr, "Failed to get cq_event\n");
         return 1;
 }
  /* Ack the event */
 ibv_ack_cq_events(ev_cq, 1);
```

Protection domain management

Lists the functions to be used for managing a protection domain for the `libibverb` library.

ibv_alloc_pd, ibv_dealloc_pd

Allocates or deallocates a protection domain (PD).

Syntax

```
#include <rdma/verbs.h>
struct ibv_pd *ibv_alloc_pd(struct ibv_context *context)
int ibv_dealloc_pd(struct ibv_pd *pd)
```

Description

The **ibv_alloc_pd()** function allocates a PD for the remote device memory access (RDMA) device context, the *context* parameter. The **ibv_dealloc_pd()** function deallocates PD, the *pd* parameter.

Note: The **ibv_dealloc_pd()** function fails if any other RDMA resource is still associated with the PD that must be freed.

Parameters

Item	Descriptor
context	The **ibv_context** struct for the **ibv_open_device()** function.

Return Value

The **ibv_alloc_pd()** function returns a pointer to the allocated PD, or NULL if the request fails. The **ibv_dealloc_pd()** function returns 0 on success, or the value of **errno** on failure (which indicates the reason for failure).

Memory region management

Lists the functions to be used for memory region management for the `libibverb` library.

ibv_reg_mr

Registers or releases a memory region (MR).

Syntax

```
#include <rdma/verbs.h>
struct ibv_mr *ibv_reg_mr(struct ibv_pd *pd, void *addr,size_t length,int ibv_access_flags
access);
int ibv_dereg_mr(struct ibv_mr *mr);
```

Description

The **ibv_reg_mr()** function registers a memory region (MR) that is associated with the protection domain, the *pd* parameter. The starting address of the MR is specified by using the *addr* parameter and its size is specified by using the *length* parameter. The *access* parameter describes the required memory protection attributes that are either 0 or the bitwise OR of one or more of the following flags:

IBV_ACCESS_LOCAL_WRITE
Enables local write access

IBV_ACCESS_REMOTE_WRITE
Enable remote write access

IBV_ACCESS_REMOTE_READ
Enable remote read access

IBV_ACCESS_REMOTE_ATOMIC
Enable remote atomic operation access (not supported)

IBV_ACCESS_MW_BIND
Enable memory window binding (not supported)

If the IBV_ACCESS_REMOTE_WRITE or IBV_ACCESS_REMOTE_ATOMIC flag is set, the IBV_ACCESS_LOCAL_WRITE flag must also be set.

Note: Local read access is always enabled for the MR.

The **ibv_dereg_mr()** function release the MR.

Parameters

Item	Descriptor
pd	Specifies the **ibv_pd** struct for the **ibv_alloc_pd()** function.
addr	Specifies the memory base address.
length	Specifies the length of memory region in bytes.
access	Specifies the access flags.

Return Values

The **ibv_reg_mr()** function returns a pointer to the registered MR on success, and NULL if the request fails. The local key (L_Key) *lkey* field is used by the **ibv_sge** struct when posting buffers with **ibv_post_*** verbs, and the remote key (R_Key) *rkey* field is used by remote processes to run the remote device memory access (RDMA) operations. The remote process places the *rkey* field in the **ibv_send_wr** struct that is sent to the **ibv_post_send()** function.

The **ibv_dereg_mr()** function returns 0 on success, and the value of **errno** on failure, which indicates the reason for failure.

Event Management

Lists the functions that is used to manage an event for the `libibverbs` library.

ibv_create_comp_channel, ibv_destroy_comp_channel

Creates or destroys a completion event channel.

Syntax

```
#include <rdma/verbs.h>
struct ibv_comp_channel *ibv_create_comp_channel(struct ibv_context *context)
int ibv_destroy_comp_channel(struct ibv_comp_channel *channel)
```

Description

The **ibv_create_comp_channel()** function creates a completion event channel for the remote direct memory access (RDMA) device context, the *context* parameter. A completion channel is a mechanism to receive notifications when a new completion queue event (CQE) is placed on a completion queue (CQ).

The **ibv_destroy_comp_channel()** function destroys the completion event channe.

Notes:

- A **completion channel** is an abstraction introduced by the **libibverbs** library that does not exist in the InfiniBand architecture verbs specification. A completion channel is essentially a file descriptor that is

used to deliver completion notifications to a userspace process. When a completion event is generated for a completion queue (CQ), the event is delivered through the completion channel attached to that CQ. This process might be useful to send completion events to different threads by using multiple completion channels.

- The **ibv_destroy_comp_channel()** function fails if any CQs are still associated with the completion event channel that is being destroyed.

Parameters

Item	Descriptor
context	The **ibv_context** struct for the **ibv_open_device()** function.

Return Value

The **ibv_create_comp_channel()** function returns a pointer to the created completion event channel, or NULL if the request fails.

The **ibv_destroy_comp_channel()** function returns 0 on success, or the value of **errno** on failure (which indicates the reason for failure).

ibv_get_async_event, ibv_ack_async_event

Gets or acknowledges the asynchronous events.

Syntax

```
#include <rdma/verbs.h>
int ibv_get_async_event(struct ibv_context *context,
struct ibv_async_event *event);void ibv_ack_async_event
(struct ibv_async_event *event);
```

Description

The **ibv_get_async_event()** function waits for the next async event of the remote direct memory access (RDMA) device context and returns it through the*event* pointer, which is an **ibv_async_event** struct, as defined in the <rdma/verbs.h> file.

```
struct ibv_async_event {
        union {
                struct ibv_cq    *cq;               /* CQ that got the event */
                struct ibv_qp    *qp;               /* QP that got the event */
                struct ibv_srq    *srq;        /* SRQ that got the event(Not Supported)*/
                int        port_num;                  /* port number that got the event */
        } element;
        enum ibv_event_type     event_type;     /* type of the event */
};
```

The **ibv_create_qp()** function updates the *qp_init_attr* parameter in the **cap** struct with the actual QP values of the QP that was created. The values are greater than or equal to the values requested. The **ibv_destroy_qp()** function destroys the QP by using the *qp* parameter.

One member of the element union is valid, depending on the **event_type** member of the structure. The **event_type** member can be one of the following events:

Item	Descriptor
QP events	
IBV_EVENT_QP_FATAL	Error occurred on a QP and it transitions to error state.
IBV_EVENT_QP_REQ_ERR	Invalid request that causes a local work queue error.
IBV_EVENT_QP_ACCESS_ERR	Local access violation error.
IBV_EVENT_COMM_EST	Communication is established on a QP.

Item	Descriptor
IBV_EVENT_SQ_DRAINED	Send queue is drained of outstanding messages in progress.
IBV_EVENT_PATH_MIG	A connection is moved to an alternative path.
IBV_EVENT_PATH_MIG_ERR	A connection failed to moved to the alternative path.
CQ events	
IBV_EVENT_CQ_ERR	CQ is in error (CQ overrun).
Port events	
IBV_EVENT_PORT_ACTIVE	Link became active on a port.
IBV_EVENT_PORT_ERR	Link became unavailable on a port.
IBV_EVENT_LID_CHANGE	Link ID (LID) is changed on a port.
IBV_EVENT_PKEY_CHANGE	The **P_Key** table is changed on a port.
CA events	
IBV_EVENT_DEVICE_FATAL	CA is in **FATAL** state.

The **ibv_ack_async_event()** function acknowledges the asynchronous event.

Notes:

- All asynchronous events that the **ibv_get_async_event()** function returns must be acknowledged by using the **ibv_ack_async_event()** event. To avoid competition, destroying an object (CQ or QP) waits for all affiliated events for the object to be acknowledged. This process avoids an application from retrieving an affiliated event after the corresponding object is destroyed.
- The **ibv_get_async_event()** function is a blocking function. If multiple threads call this function simultaneously, then when an async event occurs, only one thread will receive this function. It is not possible to predict the thread that receives the function.

Input Data

Item	Descriptor
`struct ibv_context *`*context*	The **ibv_context** struct for the **ibv_open_device** function.
`struct ibv_async_event *`*event*	The event pointer.

Return Value

The **ibv_get_async_event()** function returns 0 on success, and -1 if the request fails.

The **ibv_ack_async_event()** function returns no value.

Example

The following code example demonstrates one possible way to work with async events in nonblocking mode. The event executes the following steps:

1. Sets the async events queue in the nonblocked work mode.
2. Polls the queue until it has an asynchronous event.
3. Gets the asynchronous event and acknowledges it.

```
/* change the blocking mode of the async event queue */
flags = fcntl(ctx->async_fd, F_GETFL);
rc = fcntl(ctx->async_fd, F_SETFL, flags | O_NONBLOCK);
if (rc &lt; 0) {
       fprintf(stderr, "Failed to change file descriptor of async event queue\n");
       return 1;
}
/*
 * poll the queue until it has an event and sleep ms_timeout
 * milliseconds between any iteration
```

```
 */
my_pollfd.fd      = ctx->async_fd;
my_pollfd.events  = POLLIN;
my_pollfd.revents = 0;

do {
        rc = poll(&my_pollfd;,1, ms_timeout);
} while (rc == 0);
if (rc < 0) {
                        fprintf(stderr, "poll failed\n");
                return 1;
 }

/* Get the async event */
if (ibv_get_async_event(ctx, &async_event)) {
         fprintf(stderr, "Failed to get async_event\n");
         return 1;
 }
 /* Ack the event */
ibv_ack_async_event(&async_event);
```

ibv_event_type_str()

Returns the string that describes the **event_type**, **node_type**, and **port_state** enum values.

Syntax

```
const char *ibv_event_type_str(enum ibv_event_type event_type);
const char *ibv_node_type_str(enum ibv_node_type node_type);
const char *ibv_port_state_str(enum ibv_port_state port_state);
```

Description

The **ibv_node_type_str()** function returns a string that describes the *node_type* enum value.

The **ibv_port_state_str()** function returns a string that describes the *port_state* enum value.

The **ibv_event_type_str()** function returns a string that describes the *event_type* enum value.

Return Value

The **ibv_node_type_str()**, **ibv_port_state_str()**, and **ibv_event_type_str()** functions return a constant string that describes the enum value passed as their argument.

The <<unknown>> string is passed if the enum value is not known.

Verbs not supported by the libibverbs library

You can find the list of verbs that are not supported by the **libibverbs** library.

Item	Descriptor
Shared Receive Queues (SRQ)	
ibv_create_srq	Creates a shared receive queue.
ibv_modify_srq	Modifies attributes of a shared receive queue.
ibv_query_srq	Gets the attributes of a shared receive queue.
ibv_destroy_srq	Destroys a shared receive queue.
ibv_post_srq_recv	Posts a list of work requests to a shared receive queue.
Extended Reliable Connection (XRC)	
ibv_create_xrc_srq	Creates an XRC shared receive queue
ibv_open_xrc_domain	Opens an eXtended Reliable Connection domain.

Item	Descriptor
ibv_close_xrc_domain	Closes an eXtended Reliable Connection domain.
ibv_create_xrc_rcv_qp	Creates an XRC queue pair for serving as a receive-side only queue pair (QP).
ibv_modify_xrc_rcv_qp	Modifies the attributes of an XRC receive QP.
ibv_query_xrc_rcv_qp	Gets the attributes of an XRC receive QP.
ibv_reg_xrc_rcv_qp	Registers a user process with an XRC receive QP.
ibv_unreg_xrc_rcv_qp	Unregister a user process with an XRC receive QP.
ibv_fork_init	Initializes the libibverbs library to support the fork () function.

dlcclose Entry Point of the GDLC Device Manager

Purpose

Closes a generic data link control (GDLC) channel.

Syntax

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

int dlcclose (*devno*, *chan*)

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being closed.

Description

Each GDLC supports the **dlcclose** entry point as its switch table entry for the **close** subroutine. The file system calls this entry point from the process environment only.The **dlcclose** entry point is called when a user's application program invokes the **close** subroutine or when a kernel user calls the **fp_close** kernel service. This routine disables a GDLC channel for the user. If this is the last channel to close on the port, the GDLC device manager issues a close to the network device handler and deletes the kernel process that serviced device handler events on behalf of the user.

Parameters

Item	Description
devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
chan	Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.

Return Values

Item	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number. This value is defined in the **/usr/include/sys/errno.h** file.

dlcconfig Entry Point of the GDLC Device Manager

Purpose

Configures the generic data link control (GDLC) device manager.

Syntax

```
#include <sys/uio.h>
#include <sys/device.h>
```

int dlcconfig (*devno*, *op*, *uiop*)

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being configured.

Description

The **dlcconfig** entry point is called during the kernel startup procedures to initialize the GDLC device manager with its device information. The operating system also calls this routine when the GDLC is being terminated or queried for vital product data.

Each GDLC supports the **dlcconfig** entry point as its switch table entry for the **sysconfig** subroutine. The file system calls this entry point from the process environment only.

Parameters

Item	Description
devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. One **dev_t** device number exists for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
op	Specifies the operation code that indicates the function to be performed: **CFG_INIT** Initializes the GDLC device manager. **CFG_TERM** Terminates the GDLC device manager. **CFG_QVPD** Queries GDLC vital product data. This operation code is optional.
uiop	Points to the **uio** structure specifying the location and length of the caller's data area for the **CFG_INIT** and **CFG_QVPD** operation codes. No data areas are specifically defined for GDLC, but DLCs can define the data areas for a particular network.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file:

Item	Description
0	Indicates a successful operation.
EINVAL	Indicates an invalid value.
ENODEV	Indicates that no such device handler is present.
EFAULT	Indicates that a kernel service, such as the **uiomove** or **devswadd** kernel service, has failed.

dlcioctl Entry Point of the GDLC Device Manager

Purpose

Issues specific commands to generic data link control (GDLC).

Syntax

```
#include <sys/device.h>
#include <sys/gdlextcb.h>
```

```
int dlcioctl (devno, op, arg, devflag, chan, ext)
```

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being controlled.

Description

The **dlcioctl** entry point is called when an application program invokes the **ioctl** subroutine or when a kernel user calls the **fp_ioctl** kernel service. The **dlcioctl** routine decodes commands for special functions in the GDLC.

Each GDLC supports the **dlcioctl** entry point as its switch table entry for the **ioctl** subroutine. The file system calls this entry point from the process environment only.

Parameters

Item	Description
devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. One **dev_t** device number exists for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
op	Specifies the parameter from the subroutine that specifies the operation to be performed.
arg	Indicates the parameter from the subroutine that specifies the address of a parameter block.
devflag	Specifies the flag word with the following flags defined: **DKERNEL** Entry point called by kernel routine using the **fp_open** kernel service. This indicates that the *arg* parameter points to kernel space. **DREAD** Open for reading. This flag is ignored. **DWRITE** Open for writing. This flag is ignored. **DAPPEND** Open for appending. This flag is ignored. **DNDELAY** Device open in nonblocking mode. This flag is ignored.
chan	Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.
ext	Specifies the extended subroutine parameter. This parameter is ignored by GDLC.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file.

Value	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number.
EINVAL	Indicates an invalid value.
ENOMEM	Indicates insufficient resources to satisfy the **ioctl** subroutine.

dlcmpx Entry Point of the GDLC Device Manager

Purpose

Decodes the device handler's special file name appended to the open call.

Syntax

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

int dlcmpx ( devno, chanp,  channame)
```

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being opened.

Description

The operating system calls the **dlcmpx** entry point when a generic data link control (GDLC) channel is allocated. This routine decodes the name of the device handler appended to the end of the GDLC special file name at open time. GDLC allocates the channel and returns the value in the *chanp* parameter.

This routine is also called following a **close** subroutine to deallocate the channel. In this case the *chanp* parameter is passed to GDLC to identify the channel being deallocated. Since GDLC allocates a new channel for each **open** subroutine, a **dlcmpx** routine follows each call to the **dlcclose** routine.

Each GDLC supports the **dlcmpx** entry point as its switch table entry for the **open** and **close** subroutines. The file system calls this entry point from the process environment only.

Parameters

Item	Description
devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. There is one **dev_t** device number for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
chanp	Specifies the channel ID returned if a valid path name exists for the device handler, and the **openflag** is set. If no channel ID is allocated, this parameter is set to a value of -1 by GDLC.
channame	Points to the appended path name (path name extension) of the device handler that is used by GDLC to attach to the network. If this is null, the channel is deallocated.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file:

Value	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number.
EINVAL	Indicates an invalid value.

dlcopen Entry Point of the GDLC Device Manager

Purpose

Opens a generic data link control (GDLC) channel.

Syntax

```
#include <sys/device.h>
#include <sys/gdlextcb.h>
```

int dlcopen (*devno*, *devflag*, *chan*, *ext***)**

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being opened.

Description

The **dlcopen** entry point is called when a user's application program invokes the **open** or **openx** subroutine, or when a kernel user calls the **fp_open** kernel service. The GDLC device manager opens the specified communications device handler and creates a kernel process to catch posted events from that port. Additional opens to the same port share both the device handler open and the GDLC kernel process created on the original open.

Each GDLC supports the **dlcopen** entry point as its switch table entry for the **open** and **openx** subroutines. The file system calls this entry point from the process environment only.

Note: It may be more advantageous to handle the actual device handler open and kernel process creation in the **dlcmpx** routine. This is left as a specific DLC's option.

Parameters

Item	Description
devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. One **dev_t** device number exists for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.

Item	Description
devflag	Specifies the flag word with the following flags defined: **DKERNEL** Entry point called by kernel routine using the **fp_open** kernel service. All command extensions and **ioctl** arguments are in kernel space. **DREAD** Open for reading. This flag is ignored. **DWRITE** Open for writing. This flag is ignored. **DAPPEND** Open for appending. This flag is ignored. **DNDELAY** Device open in nonblocking mode. This flag is ignored.
chan	Specifies the channel ID assigned by GDLC in the **dlcmpx** routine.
ext	Specifies the extended subroutine parameter. This is a pointer to the **dlc_open_ext** extended I/O structure for the **open** subroutine.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file.

Value	Description
0	Indicates a successful operation.
ECHILD	Indicates that the device manager cannot create a kernel process.
EINVAL	Indicates an invalid value.
ENODEV	Indicates that no such device handler is present.
ENOMEM	Indicates insufficient resources to satisfy the **open** subroutine.
EFAULT	Indicates that a kernel service, such as the **copyin** or **initp** kernel service was unsuccessful.

dlcread Entry Point of the GDLC Device Manager

Purpose

Reads receive data from generic data link control (GDLC).

Syntax

```
#include <sys/device.h>
#include <sys/gdlextcb.h>
```

```
int dlcread (devno, uiop, chan, ext)
```

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being read.

Description

The **dlcread** entry point is called when a user application program invokes the **readx** subroutine. Kernel users do *not* call an **fp_read** kernel service. All receive data is returned to the user in the same order as

received. The type of data that was read is indicated, as well as the service access point (SAP) and link station (LS) identifiers.

The following fields in the **uio** and **iov** structures are used to control the read-data transfer operation:

Field	Description
`uio_iov`	Points to an **iovec** structure.
`uio_iovcnt`	Indicates the number of elements in the **iovec** structure. This must be set to a value of 1. Vectored read operations are not supported.
`uio_offset`	Indicates the file offset established by a previous **fp_lseek** kernel service. This field is ignored by GDLC.
`uio_segflag`	Indicates whether the data area is in application or kernel space. This is set to the **UIO_USERSPACE** value by the file I/O subsystem to indicate application space.
`uio_fmode`	Contains the value of the file mode set with the **open** applications subroutine to GDLC.
`uio_resid`	Specifies initially the total byte count of the receive data area. GDLC decrements this count for each packet byte received using the **uiomove** kernel service.
`iovec structure`	Contains the starting address and length of the received data.
`iov_base`	Specifies where GDLC writes the address of the received data. This field is a variable in the **iovec** structure.
`iov_len`	Contains the byte length of the data. This field is a variable in the **iovec** structure.

Each GDLC supports the **dlcread** entry point as its switch table entry for the **readx** subroutine. The file system calls this entry point from the process environment only.

Parameters

Item	Description
devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. One **dev_t** device number exists for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
uiop	Points to the **uio** structure containing the read parameters.
chan	Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.
ext	Specifies the extended subroutine parameter. This is a pointer to the extended I/O structure. The argument to this parameter must always be in the application space.

Return Values

Successful read operations and those truncated due to limited user data space each return a value of 0 (zero). If more data is received from the media than will fit into the application data area, the **DLC_OFLO** value indicator is set in the command extension area (**dlc_io_ext**) to indicate that the read is truncated. All excess data is lost.

The following return values are defined in the **/usr/include/sys/errno.h** file:

Value	Description
EBADF	Indicates a bad file number.

Value	Description
EINTR	Indicates that a signal interrupted the routine before it received data.
EINVAL	Indicates an invalid value.
ENOMEM	Indicates insufficient resources to satisfy the read operation.

dlcselect Entry Point of the GDLC Device Manager

Purpose

Selects for asynchronous criteria from generic data link control (GDLC), such as receive data completion and exception conditions.

Syntax

```
#include <sys/device.h>
#include <sys/poll.h>
#include <sys/gdlextcb.h>
```

```
int dlcselect (devno, events, reventp, chan)
```

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being selected.

Description

The **dlcselect** entry point is called when a user application program invokes a **select** or **poll** subroutine. This allows the user to select receive data or exception conditions. The **POLLOUT** write-availability criteria is not supported. If no results are available at the time of a **select** subroutine, the user process is put to sleep until an event occurs.

If one or more events specified in the *events* parameter are true, the **dlcselect** routine updates the *reventp* (returned events) parameter (passed by reference) by setting the corresponding event bits that indicate which events are currently true.

If none of the requested events are true, the **dlcselect** routine sets the returned events parameter to a value of 0 (passed by reference using the *reventp* parameter) and checks the **POLLSYNC** flag in the *events* parameter. If this flag is true, the routine returns because the event request was a synchronous request. If the **POLLSYNC** flag is false, an internal flag is set for each event requested in the *events* parameter.

When one or more of the requested events become true, GDLC issues the **selnotify** kernel service to notify the kernel that a requested event or events have become true. The internal flag indicating that the event was requested is then reset to prevent renotification of the event.

If the port in use is in a closed state, implying that the requested event or events can never be satisfied, GDLC sets the returned events flags to a value of 1 for each event that can never be satisfied. This is done so that the **select** or **poll** subroutine does not wait indefinitely.

Kernel users do not call an **fp_select** kernel service since their receive data and exception notification functions are called directly by GDLC.

Each GDLC supports the **dlcselect** entry point as its switch table entry for the **select** or **poll** subroutines. The file system calls this entry point from the process environment only.

Parameters

Item	Description
devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. One **dev_t** device number exists for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
events	Identifies the events to check. The following events are: **POLLIN** Read selection. **POLLOUT** Write selection. This is not supported by GDLC. **POLLPRI** Exception selection. **POLLSYNC** This request is a synchronous request only. The routine should not perform a **selnotify** kernel service routine due to this request if the events occur later.
reventp	Identifies a returned events pointer. This is a parameter passed by reference to indicate which of the selected events are true at the time of the call. See the preceding *events* parameter for possible values.
chan	Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file:

Value	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the subroutine before it found any of the selected events.
EINVAL	Indicates that the specified **POLLOUT** write selection is not supported.

dlcwrite Entry Point of the GDLC Device Manager

Purpose

Writes transmit data to generic data link control (GDLC).

Syntax

```
#include <sys/uio.h>
#include <sys/device.h>
#include <sys/gdlextcb.h>
```

```
int dlcwrite (devno, uiop, chan, ext)
```

Note: The **dlc** prefix is replaced with the three-digit prefix for the specific GDLC device manager being written.

Description

The **dlcwrite** entry point is called when a user application program invokes a **writex** subroutine or when a kernel user calls the **fp_write** kernel service. An extended write is used in order to specify the type of data being sent, as well as the service access point (SAP) and link station (LS) identifiers.

The following fields in the **uio** and **iov** structures are used to control the write data transfer operation:

Field	Description
`uio_iov`	Points to an **iovec** structure.
`uio_iovcnt`	Indicates the number of elements in the **iovec** structure. This must be set to a value of 1 for the kernel user, indicating that there is a single communications memory buffer (**mbuf**) chain associated with the **write** subroutine.
`uio_offset`	Specifies the file offset established by a previous **fp_lseek** kernel service. This field is ignored by GDLC.
`uio_segflag`	Indicates whether the data area is in application or kernel space. This field is set to the **UIO_USERSPACE** value by the file I/O subsystem if the data area is in application space. The field must be set to the **UIO_SYSSPACE** value by the kernel user to indicate kernel space.
`uio_fmode`	Contains the value of the file mode set during an application **open** subroutine to GDLC or can be set directly during a **fp_open** kernel service to GDLC.
`uio_resid`	Contains the total byte count of the transmit data area for application users. For kernel users, GDLC ignores this field since the communications memory buffer (**mbuf**) also carries this information.
`iovec structure`	Contains the starting address and length of the transmit.
`iov_base`	Specifies a variable in the **iovec** structure where GDLC gets the address of the application user's transmit data area or the address of the kernel user's transmit **mbuf**.
`iov_len`	Specifies a variable in the **iovec** structure that contains the byte length of the application user's transmit data area. This variable is ignored by GDLC for kernel users, since the transmit **mbuf** contains a length field.

Each GDLC supports the **dlcwrite** entry point as its switch table entry for the **writex** subroutine. The file system calls this entry point from the process environment only.

Parameters

Item	Description
devno	Indicates major and minor device numbers. This is a **dev_t** device number that specifies both the major and minor device numbers of the GDLC device manager. One **dev_t** device number exists for each type of GDLC, such as Ethernet, Token-Ring, or SDLC.
uiop	Points to the **uio** structure containing the write parameters.
chan	Specifies the channel ID assigned by GDLC in the **dlcmpx** routine at open time.
ext	Specifies the extended subroutine parameter. This is a pointer to the extended I/O structure. This data must be in the application space if the `uio_fmode` field indicates an application subroutine or in the kernel space if the `uio_fmode` field indicates a kernel subroutine.

Return Values

The following return values are defined in the **/usr/include/sys/errno.h** file:

Value	Description
0	Indicates a successful operation.
EAGAIN	Indicates that transmit is temporarily blocked and a sleep cannot be issued.
EBADF	Indicates a bad file number (application).
EINTR	Indicates that a signal interrupted the routine before it could complete successfully.
EINVAL	Indicates an invalid value, such as too much data for a single packet.
ENOMEM	Indicates insufficient resources to satisfy the **write** subroutine, such as a lack of communications memory buffers (**mbufs**).
ENXIO	Indicates an invalid file pointer (kernel).

Exception Condition Routine for DLC

Purpose

Notifies the kernel user each time an asynchronous event occurs in generic data link control (GDLC).

Syntax

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

int (*dlc_open_ext.excp_fa)(*ext***)**
struct dlc_getx_arg ****ext*;**

Description

The DLC Exception Condition routine notifies the kernel user each time an asynchronous event occurs, such as **DLC_SAPD_RES** (SAP-disabled) or **DLC_CONT_RES** (contacted), in GDLC.

Each GDLC supports a subset of the data-received routines. It is critical to performance that the Exception Condition routine for DLC be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
ext	Specifies the same structure for a **dlc_getx_arg** (get exception) ioctl subroutine.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the exception has been accepted.

Note: The function call above has a hidden parameter extension for internal use only, defined as **int ****chanp*, the channel pointer.

I-Frame Data Received Routine for DLC

Purpose

Receives a normal sequenced data packet each time it is coded by the kernel user and called by generic data link control (GDLC).

Syntax

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

int (*dlc_open_ext.rcvi_fa)( m,  ext)
struct mbuf *m;
struct dlc_io_ext *ext;
```

Description

The DLC I-Frame Data Received routine receives a normal sequenced data packet each time it is coded by the kernel user and called by GDLC.

Each GDLC supports a subset of the data-received routines. It is critical to performance that the I-Frame Data Received routine be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
m	Points to a communications memory buffer (**mbuf**).
ext	Specifies the receive extension parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for reads. The argument to this parameter must be in the kernel space.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the received **I-frame** function call is accepted.
DLC_FUNC_BUSY	Indicates that the received **I-frame** function call cannot be accepted at this time. The ioctl command operation **DLC_EXIT_LBUSY** must be issued later using the **ioctl** subroutine.
DLC_FUNC_RETRY	Indicates that the received **I-frame** function call cannot be accepted at this time. GDLC should retry this function call later. The actual retry wait period depends on the DLC in use. Excessive retries can be subject to a halt of the link station.

ioctl Operations (op) for DLC

Syntax

```
#define  DLC_ENABLE_SAP      1
#define  DLC_DISABLE_SAP     2
#define  DLC_START_LS        3
#define  DLC_HALT_LS         4
#define  DLC_TRACE           5
```

```
#define  DLC_CONTACT            6
#define  DLC_TEST               7
#define  DLC_ALTER              8
#define  DLC_QUERY_SAP          9
#define  DLC_QUERY_LS          10
#define  DLC_ENTER_LBUSY       11
#define  DLC_EXIT_LBUSY        12
#define  DLC_ENTER_SHOLD       13
#define  DLC_EXIT_SHOLD        14
#define  DLC_GET_EXCEP         15
#define  DLC_ADD_GRP           16
#define  DLC_ADD_FUNC_ADDR     17
#define  DLC_DEL_FUNC_ADDR     18
#define  DLC_DEL_GRP           19
#define  IOCINFO      /* see /usr/include/sys/ioctl.h */
```

Description

Note: If the operation's notification is returned asynchronously to the user by way of exception, application users should refer to **DLC_GET_EXCEP ioctl** operation for DLC and kernel users should refer to Exception Condition Routine for DLC for more information.

Each GDLC supports a subset of ioctl subroutine operations. These ioctl operations are selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. They may be called from the process environment only.

The following ioctl command operations are supported for generic data link control (GDLC):

Operation	Description
DLC_ADD_FUNC_ADDR	Adds a group or multicast receive functional address to a port. This command allows additional functional address bits to be added to the current receive functional address mask, as supported by the individual device handlers. See device handler specifications to determine which address values are supported. **Note:** Currently, token ring is the only local area network (LAN) protocol supporting functional addresses.
DLC_ADD_GRP	Adds a group or multicast receive address to a port. This command allows additional address values to be filtered in receive as supported by the individual communication device handlers. See device handler specifications to determine which address values are supported.
DLC_ALTER	Alters link station (LS) configuration.
DLC_CONTACT	Contacts the remote LS. This ioctl operation does not complete processing before returning to the user. The **DLC_CONTACT** notification is returned asynchronously to the user by way of exception.
DLC_DEL_GRP	Removes a group or multicast address that was previously added to a port with a **DLC_ENABLE_SAP** or **DLC_ADD_GRP** ioctl operation.
DLC_DEL_FUNC_ADDR	Removes a group or multicast receive functional address from a port. This command removes functional address bits from the current receive functional address mask, as supported by the individual device handlers. See device handler specifications to determine which address values are supported. **Note:** Currently, token ring is the only local area network protocol supporting functional addresses.

Operation	Description
DLC_DISABLE_SAP	Disables a service access point (SAP). This ioctl operation does not fully complete the disable SAP processing before returning to the user. The **DLC_DISABLE_SAP** notification is returned asynchronously to the user later by way of exception.
DLC_ENABLE_SAP	Enables an SAP. This ioctl operation does not fully complete the enable SAP processing before returning to the user. The **DLC_ENABLE_SAP** notification is returned asynchronously to the user later by way of exception.
DLC_ENTER_LBUSY	Enters local busy mode on an LS.
DLC_ENTER_SHOLD	Enters short hold mode on an LS.
DLC_EXIT_LBUSY	Exits local busy mode on an LS.
DLC_EXIT_SHOLD	Exits short hold mode on an LS.
DLC_GET_EXCEP	Returns asynchronous exception notifications to the application user. **Note:** This ioctl command operation is not used by the kernel user since all exception conditions are passed to the kernel user by their exception handler routine.
DLC_HALT_LS	Halts an LS. This ioctl operation does not complete processing before returning to the user. Notification of the ioctl operation, **DLC_HALT_LS**, is returned asynchronously to the user by way of exception.
DLC_QUERY_LS	Queries an LS.
DLC_QUERY_SAP	Queries an SAP.
DLC_START_LS	Starts an LS. This ioctl operation does not complete processing before returning to the user. Notification of the ioctl operation, **DLC_START_LS**, is returned asynchronously to the user by way of exception.
DLC_TEST	Tests LS connectivity. This ioctl operation does not complete processing before returning to the user. Notification of the ioctl operation, **DLC_TEST** completion, is returned asynchronously to the user by way of exception.
DLC_TRACE	Traces LS activity.
IOCINFO	Returns a structure that describes the device. Refer to the description of the **/usr/include/sys/devinfo.h** file. The first byte is set to an ioctype of **DD_DLC**. The subtype and data are defined by the individual DLC devices.

ioctl Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Transfers special commands to generic data link control (GDLC) using a file descriptor.

Syntax

```
#include <sys/ioctl.h>
#include <sys/devinfo.h>
#include <sys/gdlextcb.h>
```

int ioctl (*fildes*, *op*, *arg*);

Description

The **ioctl** subroutine initiates various GDLC functions, such as changing configuration parameters, contacting a remote link, and testing a link. Most of these operations can be completed before returning to the user (synchronously). Since some operations take longer, asynchronous results are returned later using the exception condition notification. Application users can obtain these exceptions using the **DLC_GET_EXCEP** ioctl operation. For more information on the functions that can be initiated using the **ioctl** subroutine.

Each GDLC supports the **ioctl** subroutine interface via its **dlcioctl** entry point. This subroutine may be called from the process environment only.

Parameters

Item	Description
fildes	Specifies the file descriptor of the target GDLC.
op	Specifies the operation to be performed by GDLC.
arg	Specifies the address of the parameter block.

Return Values

Item	Description
0	Indicates a successful operation.

If an error occurs, a value of -1 is returned with one of the following error values available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file:

Value	Description
EBADF	Indicates a bad file number.
EINVAL	Indicates an invalid argument.
ENOMEM	Indicates insufficient resources to satisfy the **ioctl** subroutine.

Network Data Received Routine for DLC

Purpose

Receives network-specific data each time it is coded by the kernel user and called by generic data link control (GDLC).

Syntax

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

int (*dlc_open_ext.rcvn_fa)( m,  ext)
struct mbuf *m;
struct dlc_io_ext *ext;
```

Description

The DLC Network Data Received routine receives network-specific data each time the routine is coded by the kernel user and called by GDLC.

Each GDLC supports a subset of the data-received routines. It is critical to performance that the Network Data Received routine be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
m	Points to a communications memory buffer (**mbuf**).
ext	Specifies the receive extension parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for read operations.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the received network **mbuf** data has been accepted.
DLC_FUNC_RETRY	Indicates that the received network **mbuf** data cannot be accepted at this time. GDLC should retry this function call later. The actual retry wait period depends on the DLC in use. Excessive retries can cause a disabling of the service access point.

open Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Opens the generic data link control (GDLC) device manager by special file name.

Syntax

```
#include <fcntl.h>
#include <sys/gdlextcb.h>
```

int open (*path*, *oflag*, *mode***)**

```
or
```

int openx (*path*, *oflag*, *mode*, *ext***)**

Description

The **open** subroutine allows the application user to open a GDLC device manager by specifying the DLC special file name and the target device handler special file name. Since the GDLC device manager is multiplexed, more than one process can open it (or the same process many times) and still have unique channel identifications.

Each open carries the communications device handler's special file name so that the DLC knows on which port to transfer data. This name must directly follow the DLC's special file name. For example, in the `/dev/dlcether/ent0` character string, `ent0` is the special file name of the Ethernet device handler. GDLC obtains this name using its **dlcmpx** routine.

Each GDLC supports the **open** subroutine interface by way of its **dlcopen** and **dlcmpx** entry points. This subroutine may be called from the process environment only.

Parameters

Item	Description
path	Consists of a character string containing the **/dev** special file name of the GDLC device manager, with the name of the communications device handler appended as follows: `/dev/dlcether/ent0`
oflag	Specifies a value for the file status flag. The GDLC device manager ignores all but the following flags: **O_RDWR** Open for reading and writing. This must be set for GDLC or the open will fail. **O_NDELAY, O_NONBLOCK** Subsequent reads with no data present and writes that cannot get enough resources will return immediately. The calling process is not put to sleep.
mode	Specifies the **O_CREAT** mode parameter. This is ignored by GDLC.
ext	Specifies the extended subroutine parameter. This is a pointer to the **dlc_open_ext** extended I/O structure for the open subroutines.

Return Values

Upon successful completion, the **open** subroutine returns a valid file descriptor that identifies the opened GDLC channel.

If an error occurs, a value of -1 is returned with one of the following error values available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file:

Value	Description
ECHILD	Indicates that the device manager cannot create a kernel process.
EINVAL	Indicates an invalid value.
ENODEV	Indicates that no such device handler is present.
ENOMEM	Indicates insufficient resources to satisfy the **open** subroutine.
EFAULT	Indicates that a kernel service, such as the **copyin** or **initp** kernel service, has failed.

open Subroutine Extended Parameters for DLC

Purpose

Alters certain defaulted parameters for an extended **open** (**openx**) subroutine.

Syntax

```
struct dlc_open_ext
{
   __ulong32_t  maxsaps;
   int (* rcvi_fa)();
   int (* rcvx_fa)();
   int (* rcvd_fa)();
   int (* rcvn_fa)();
```

```
   int (* excp_fa)();
};
```

Description

An extended **open** or **openx** subroutine can be issued to alter certain defaulted parameters, such as maximum service access points (SAPs) and ring queue depths. Kernel users may change these normally defaulted parameters, but are required to provide additional parameters to notify the **dlcopen** routine that these callers are to be treated as kernel processes and not as application processes. Additional parameters passed include functional addresses that generic data link control (GDLC) calls to notify about asynchronous events, such as receive data available.

The *maxsaps* parameter is optional for both the application and the kernel user. The other five parameters are mandatory for kernel users but are ignored by GDLC for application users. There are no default values. Each field must be filled in by the kernel user. All functional entry addresses must be valid. That is, entry points that the kernel user does not wish to support must at least point to a routine which frees the communication's memory buffer (**mbuf**) passed on the call.

These DLC extended parameters for the **open** subroutine are part of the data link control in BOS Extensions 2 for the device manager you are using.

See the **/usr/include/sys/gdlextcb.h** file for more details on GDLC structures.

Parameters

Item	Description
maxsaps	Specifies the maximum number of SAPs the user channel uses to start and run concurrently. Any value from 1 to 127 can be specified. If the default value of 1 is desired, the user must set the field to 0 (zero) before issuing the **open** subroutine.
rcvi_fa	Points to the address of a user I-Frame Data Received routine that handles the sequenced receive data completions. This field is valid for kernel users only and must be set to 0 (zero) by application users.
rcvx_fa	Points to the address of a user XID Data Received routine that handles the exchange ID receive data completions.
rcvd_fa	Points to the address of a user Datagram Data Received routine that handles the datagram receive data completions.
rcvn_fa	Points to the address of a user Network Data Received routine that handles the network receive data completions.
excp_fa	Points to the address of a user Exception Condition routine that handles the exception conditions, such as **DLC_SAPE_RES** (SAP-enabled) or **DLC_CONT_RES** (LS-contacted).

Parameter Blocks by ioctl Operation for DLC

Each command operation has a specific parameter block associated with the command pointed to by the *arg* pointer. Some parameters are sent to the generic data link control (GDLC) and others are returned. The ioctl command operations for DLC are:

read Subroutine Extended Parameters for DLC

Purpose

Provide generic data link control (GDLC) with a structure to return data types and service access point (SAP) and link station (LS) correlator.

Syntax

```
#define  DLC_INFO   0x80000000
#define  DLC_XIDD   0x40000000
#define  DLC_DGRM   0x20000000
#define  DLC_NETD   0x10000000
#define  DLC_OFLO   0x00000002
#define  DLC_RSPP   0x00000001
```

```
struct dlc_io_ext
{
   __ulong32_t  sap_corr;
   __ulong32_t  ls_corr;
   __ulong32_t  flags;
   __ulong32_t  dlh_len;
};
```

Description

An extended read or readx subroutine must be issued by an application user to provide GDLC with a structure to return the type of data and the SAP and LS correlator.

Parameters

sap_corr
Specifies the user's SAP identifier of the received data.

ls_corr
Specifies the user's LS identifier of the received data.

flags
Specifies flags for the readx subroutine. The following flags are supported:

DLC_INFO
Indicates that normal sequenced data has been received for a link station using an I-Frame Data Received routine. If buffer overflow (OFLO) is indicated, the received data has been truncated because the received data length exceeds either the maximum I-field size derived at completion of DLC_START_LS ioctl operation or the application user's buffer size.

DLC_XIDD
Indicates that exchange identification (XID) data has been received for a link station using an XID Data Received routine. If buffer overflow (OFLO) is indicated, the received XID has been truncated because the received data length exceeds either the maximum I-field size derived at DLC_START_LS completion or the application user's buffer size. If response pending (RSPP) is indicated, an XID response is required and must be provided to GDLC using a write XID as soon as possible to avoid repolling and possible termination of the remote LS.

DLC_DGRM
Indicates that a datagram has been received for an LS using a Datagram Data Received routine. If buffer overflow (OFLO) is indicated, the received data has been truncated because the received data length exceeds either the maximum I-field size derived at DLC_START_LS completion or the application user's buffer size.

DLC_NETD
Indicates that data has been received from the network for a service access point using a Network Data Received routine. This may be link-establishment data such as X.21 call-progress signals or Smartmodem command responses. It can also be data destined for the user's SAP when no link station has been started that fits the addressing of the packet received. If buffer overflow (OFLO) is indicated, the received data has been truncated because the received data length exceeds either the maximum packet size derived at DLC_ENABLE_SAP completion or the application user's buffer size.

Network data contains the entire MAC layer packet, excluding any fields stripped by the adapter such as Preamble or CRC.

DLC_OFLO
Indicates that overflow of the user data area has occurred and the data was truncated. This error does not set a u.u_error indication.

DLC_RSPP
Indicates that the XID received requires an XID response to be sent back to the remote link station.

dlh_len
Specifies data link header length. This field has a different meaning depending on whether the extension is for a readx subroutine call to GDLC or a response from GDLC.

On the application readx subroutine, this field indicates whether the user wishes to have datalink header information prefixed to the data. If this field is set to 0 (zero), the data link header is not to be copied (only the I-field is copied). If this field is set to any nonzero value, the data link header information is included in the read operation.

On the response to an application readx subroutine, this field contains the number of data link header bytes received and copied into the data link header information field.

On asynchronous receive function handlers to the kernel user, this field contains the length of the data link header within the communications memory buffer (mbuf) .

These DLC extended parameters for the read subroutine are part of the data link control in BOS Extensions 2 for the device manager you are using.

readx Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Allows receive application data to be read using a file descriptor.

Syntax

```
#include <sys/gdlextcb.h>
#include <sys/uio.h>
```

```
int readx (fildes, buf, len, ext)
```

Description

The receive queue for this application user is interrogated for any pending data. The oldest data packet is copied to user space, with the type of data, the link station correlator, and the service access point

(SAP) correlator written to the extension area. When attempting to read an empty receive data queue, the default action is to delay until data is available. If the **O_NDELAY** or **O_NONBLOCK** flags are specified in the **open** subroutine, the **readx** subroutine returns immediately to the caller.

Data is transferred using the **uiomove** kernel service between the user space and kernel communications memory buffers (**mbufs**). A complete receive packet must fit into the user's read data area. Generic data link control (GDLC) does not break up received packets into multiple user data areas.

Each GDLC supports the **readx** subroutine interface via its **dlcread** entry point. This subroutine can be called from the process environment only.

Parameters

Item	Description
fildes	Specifies the file descriptor returned from the **open** subroutine.
buf	Points to the user data area.
len	Contains the byte count of the user data area.
ext	Specifies the extended subroutine parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for the **readx** subroutine. **Note:** It is the user's responsibility to set the *ext* parameter area to 0 (zero) before issuing the **readx** subroutine to insure valid entries when no data is available.

Return Values

Upon successful completion, the **readx** subroutine returns the number of bytes read and placed into the application data area. If more data is received from the media than will fit into the application data area, the **DLC_OFLO** flag is set in the **dlc_io_ext** command extension area to indicate that the read is truncated. All excess data is lost.

If no data is available and the application user has specified the **O_NDELAY** or **O_NONBLOCK** flags at open time, a 0 (zero) is returned.

If an error occurs, a value of -1 is returned with one of the following error numbers available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file:

Value	Description
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the subroutine before it received data.
EINVAL	Indicates an invalid value.
ENOMEM	Indicates insufficient resources to satisfy the read operation.

write Subroutine Extended Parameters for DLC

Purpose

Provide generic data link control (GDLC) with data types, service access points (SAPs), and link station (LS) correlators.

Syntax

#define __DLC_INFO 0x80000000 #define __DLC_XIDD 0x40000000 #define __DLC_DGRM 0x20000000 #define __DLC_NETD 0x10000000

__ulong32_t *sap_corr*; **__ulong32_t** *ls_corr*; **__ulong32_t** *flags*; **__ulong32_t** *dlh_len*; **};**

Description

An extended **write** or **writex** subroutine must be issued by an application or kernel user to provide GDLC with data types, SAPs, and LS correlators.

These DLC extended parameters for the **write** subroutine are part of the data link control in BOS Extensions 2 for the device manager you are using.

Parameters

Item	Description
sap_corr	Specifies the GDLC SAP correlator of the write data. This field must contain the same correlator value passed back form GDLC in the `gdlc_sap_corr` field when the SAP was enabled.
dlh_len	Not used for writes.
ls_corr	Specifies the GDLC LS correlator of the write data. This field must contain the same correlator value passed back from GDLC in the `gdlc_ls_corr` field when the LS was started.

Item	Description
flags	Specifies flags for the **writex** subroutine. The following flags are supported: **DLC_INFO** Requests a sequenced data class of information to be sent (generally called I-frames). This request is valid any time the target link station has been started and contacted. **DLC_XIDD** Requests an exchange identification (XID) non-sequenced command or response packet to be sent. This request is valid any time the target link station has been started with the following rules: GDLC sends the XID as a command as long as no **DLC_TEST**, **DLC_CONTACT**, **DLC_HALT_LS**, or **DLC_XIDD** write subroutine is already in progress, and no received XID is waiting for a response. If a received XID is waiting for a response, GDLC automatically sends the write XID as that response. If no response is pending and a command is already in progress, the write is rejected by GDLC. **DLC_DGRM** Requests a datagram packet to be sent. A datagram is an unnumbered information (UI) response. This request is valid any time the target link station has been started. **DLC_NETD** Requests that network data be sent. Examples of network data include special modem control data or user-generated medium access control (MAC) and logical link control (LLC) headers. Network data must contain the entire MAC layer packet headers so that the packet can be sent without the data link control (DLC)'s intervention. GDLC only provides a pass-through function for this type of write. This request is valid any time the SAP is open.

writex Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Allows application data to be sent using a file descriptor.

Syntax

```
#include <sys/gdlextcb.h>
#include <sys/uio.h>
```

```
int writex (fildes, buf, len, ext)
```

Description

Four types of data can be sent to generic data link control (GDLC). Network data can be sent to a service access point (SAP), while normal, Exchange Identification (XID) or datagram data can be sent to a link station (LS). Data is transferred using the **uiomove** kernel service between the application user space and kernel communications I/O buffers (**mbufs**). All data must fit into a single packet for each **write** subroutine. The generic data link control does not separate the user's write data area into multiple transmit packets. A maximum write data size is passed back to the user at **DLC_ENABLE_SAP** completion and at **DLC_START_LS** completion for this purpose.

Normally, GDLC can immediately satisfy a **write** subroutine by completing the data link headers and sending the transmit packet down to the device handler. In some cases, however, transmit packets can be blocked by the particular protocol's flow control or by a resource outage. GDLC reacts to this differently, based on the system blocked or nonblocked file status flags. These are set for each channel using the **O_NDELAY** and **O_NONBLOCK** values passed on **open** or **fcntl** subroutines with the **F_SETFD** parameter.

GDLC only looks at the `uio_fmode` field on each **write** subroutine to determine whether the operation is blocked or nonblocked. Nonblocked writes that cannot get enough resources to queue the data return an error indication. Blocked **write** subroutines put the calling process to sleep until the resources free up or an error occurs.

Each GDLC supports the **writex** subroutine interface via its **dlcwrite** entry point. This subroutine may be called from the process environment only.

Note: GDLC does not support nonblocked transmit users based on resource availability using the **selwakeup** subroutine. Internal resources such as communications I/O buffers and control block locks are very dynamic. Any **write** subroutines that fail with errors (such as **EAGAIN** or **ENOMEM**) should be retried at the user's discretion.

Parameters

Item	Description
fildes	Specifies the file descriptor returned from the **open** subroutine.
buf	Points to the user data area.
len	Contains the byte count of the user data area.
ext	Specifies the extended subroutine parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for the **writex** subroutine.

Return Values

Upon successful completion, this service returns the number of bytes that were written into a communications packet from the user data area.

If an error occurs, a value of -1 is returned with one of the following error values available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file.

Value	Description
EAGAIN	Indicates insufficient resources to satisfy the write. For example, the routine was unable to obtain a necessary lock. The user can try again later.
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the subroutine before it completed successfully.
EINVAL	Indicates an invalid value, such as too much data for a single packet.
EIO	Indicates that an I/O error has occurred, such as loss of the port.
ENOMEM	Indicates insufficient resources to satisfy the write operation. For example, a lack of communications memory buffers (**mbufs**). The user can try again later.

XID Data Received Routine for DLC

Purpose

Receives an exchange identification (XID) packet each time it is coded by the kernel user and called by generic data link control (GDLC).

Syntax

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

```
int (*dlc_open_ext.rcvx_fa)( m,  ext)
struct mbuf *m;
struct dlc_io_ext *ext;
```

Description

The DLC XID Data Received routine receives an XID packet each time the routine is coded by the kernel user and called by GDLC.

Each GDLC supports a subset of the data-received routines. It is performance critical that the XID Data Received routine be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
m	Points to a communication memory buffer (**mbuf**).
ext	Specifies the receive extension parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for reads. The argument to this parameter must be in the kernel space.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the received XID **mbuf** data has been accepted.
DLC_FUNC_RETRY	Indicates that the received XID **mbuf** data cannot be accepted at this time. GDLC should retry this function call later. The actual retry wait period depends on the DLC in use. Excessive retries may close the link station.

close Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Closes the generic data link control (GDLC) device manager using a file descriptor.

Syntax

int close (*fildes***)**

Description

The **close** subroutine disables a GDLC channel. If this is the last channel to close on a port, the GDLC device manager is reset to an idle state on that port and the communications device handler is closed.

Each GDLC supports the **close** subroutine interface by way of its **dlcclose** and **dlcmpx** entry points. This subroutine can be called from the process environment only.

Parameters

Item	Description
fildes	Specifies the file descriptor of the GDLC being closed.

Return Values

Item	Description
0	Indicates a successful operation.
EBADF	Indicates a bad file number. This value is defined in the **/usr/include/sys/errno.h** file.

If an error occurs, a value of -1 is also returned.

Datagram Data Received Routine for DLC

Purpose

Receives a datagram packet each time it is coded by the kernel user and called by generic data link control (GDLC).

Syntax

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

```
int (*dlc_open_ext.rcvd_fa)( m,  ext)
struct mbuf *m;
struct dlc_io_ext *ext;
```

Description

The DLC Datagram Data Received routine receives a datagram packet each time it is coded by the kernel user and called by GDLC.

Each GDLC supports a subset of the data-received routines. It is critical to performance that the Datagram Data Received routine be coded to minimize the amount of time spent prior to returning to the GDLC that called it.

Parameters

Item	Description
m	Points to a communications memory buffer (**mbuf**).
ext	Specifies the receive extension parameter. This is a pointer to the **dlc_io_ext** extended I/O structure for read operations.

Return Values

Item	Description
DLC_FUNC_OK	Indicates that the received datagram **mbuf** data has been accepted.
DLC_FUNC_RETRY	Indicates that the received datagram **mbuf** data cannot be accepted at this time. GDLC should retry this function later. The actual retry wait period depends on the DLC in use. Excessive retries may close the link station.

DLC_DEL_GRP ioctl Operation for DLC

The **DLC_DEL_GRP** ioctl operation is selectable through the **fp_ioctl** kernel service or the **ioctl** subroutine. It can be called from the process environment only.

The following parameter removes a previously defined group or multicast address:

```
struct dlc_add_grp
   {
    __ulong32_t  gdlc_sap_corr;      /*GDLC SAP correlator  */
    __ulong32_t grpaddr_len;      /*group address length */
    uchar_t grp_addr[DLC_MAX_ADDR]; /*group address to be
                  removed        */
   };
```

The fields of this ioctl operation are:

Field	Description
`gdlc_sap_corr`	Indicates the generic data link control (GDLC) service access point (SAP) identifier being requested to remove a group or multicast address from a port. This field is known as the GDLC SAP Correlator field.
`grp_addr_len`	Contains the byte length of the group or multicast address to be removed.
`grp_addr`	Contains the group or multicast address to be removed.

select Subroutine Interface for Data Link Control (DLC) Devices

Purpose

Allows data to be sent using a file descriptor.

Syntax

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

```
int select (nfdsmsgs, readlist, writelist, exceptlist, timeout)
```

Description

The **select** subroutine checks the specified file descriptor and message queues to see if they are ready for reading (receiving) or writing (sending), or if they have an exception condition pending.

Note: Generic data link control (GDLC) does not support transmit for nonblocked notification in the full sense. If the *writelist* parameter is specified in the **select** call, GDLC always returns as if transmit is available. There is no checking to see if internal buffering is available or if internal control-block locks are free. These resources are much too dynamic, and tests for their availability can be done reasonably only at the time of use.

The *readlist* and *exceptlist* parameters are fully supported. Whenever the selection criteria specified by the *SelType* parameter is true, the file system 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 a value of 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. If the selection is not satisfied, the calling process is put to sleep waiting on a **selwakeup** subroutine at a later time.

Each GDLC supports the **select** subroutine interface via its **dlcselect** entry point. This subroutine can be called from the process environment only.

Parameters

Item	Description
nfdsmsgs	Specifies the number of file descriptors and message queues to check.
sellist	The *readlist*, *writelist*, and *exceptlist* parameters specify what to check for during reading, writing, and exceptions, respectively. Each **sellist** is a structure that contains a file descriptor bit mask (**fdsmask**) and message queue identifiers (**msgids**). The *writelist* criterion is always set to True by GDLC.
timeout	Points to a structure that specifies the maximum length of time to wait for at least one of the selection criteria to be met (if the *timeout* parameter is not a null pointer).

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 return value is similar to the

nfdsmsgs parameter in that the low-order 16 bits give the number of file descriptors. Also, the high-order 16 bits give the number of message queue identifiers. These values indicate the sum total that meet each of the read and exception criteria.

If the time limit specified by the *timeout* parameter expires, then the **select** subroutine returns a value of 0 (zero).

If an error occurs, a value of -1 is returned with one of the following error values available using the **errno** global variable, as defined in the **/usr/include/sys/errno.h** file:

Item	Description
EBADF	Indicates a bad file number.
EINTR	Indicates that a signal interrupted the subroutine before it found any of the selected events.
EINVAL	Indicates that one of the parameters contained an invalid value.

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.

© Copyright IBM Corp. 2015, 2018

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.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Index

Special Characters

_ 96
_getlong subroutine 96
_getshort subroutine 97
_ll_log subroutine 74
_putlong subroutine 97
_putshort 98
_putshort subroutine 98
/etc/hosts file
    closing 118, 119
    opening 297, 298
    retrieving host entries 149, 150, 152, 154
    setting file markers 297, 298
/etc/networks file
    closing 120, 121
    opening 301, 302
    retrieving network entries 163, 166, 167
    setting file markers 301, 302
/etc/protocols file
    closing 123
    opening 304, 305
    setting file markers 304, 305
/etc/resolv.conf file
    retrieving host entries 149, 150, 152, 154
    searching for domain names 262
    searching for Internet addresses 262
/etc/services file
    closing 124, 125
    opening 184, 305, 306
    reading 184
    retrieving service entries 180, 182
    setting file markers 305, 306
/etc/socks5c.conf File 337

A

a 99
accept subroutine 99
acknowledges asynchronous events 412
adjusting the values of entries 81
arp subroutines
    arpresolve_common 101
    arpupdate 102
arpresolve_common subroutine 101
arpupdate subroutine 102
arrays
    translating into external representations 2, 3, 25
ASCII strings
    converting to Internet addresses 217
authentication messages 13
authentication methods 295

B

b 104
bind subroutine 104
bind2addrsel subroutine 106
binds RDMA identifier 361
Booleans
    translating 2
buffers
    checking for end of file 31
byte streams
    placing long byte quantities 97

C

c 107
C language, translating
    characters 5
    discriminated unions 24
    enumerations 7
    floats 7
    integers 2, 11
    long integers 12
    numbers 27
    short integers 19
    strings 20, 26
    unsigned characters 21
    unsigned integers 21
    unsigned long integers 22, 23
call header messages 4
call messages 5
clients
    server authentication 280
close subroutine interface for DLC devices 440
closing 122
communicating with the SNMP agent 89
Communication Manager (CM) ID operations
    rdma_bind_addr 361
    rdma_connect 363
    rdma_destroy_id 360
    rdma_resolve_addr 362
    rdma_resolve_route 363
communications kernel service subroutines
    res_ninit 265
Completion queue management 405
Completion Queue management
    completion notification 406
    completion queue event 407
    ibv_get_cq_event 407
    ibv_poll_cq 406
    ibv_req_notify_cq 406
    polls a completion queue 406
compressed domain names
    expanding 112
connect 107
connect subroutine 107
connected sockets
    creating pairs 334
    receiving messages 255
    sending messages 284, 286
connecting 107

Connection Manager (CM)
ID operations 356
Connection Manager (CM) ID operations
rdma_create_ep 369
rdma_destroy_ep 370
rdma_get_dst_port 368
rdma_get_local_addr 368
rdma_get_peer_addr 368
rdma_get_src_port 367
rdma_getaddrinfo 370
rdma_migrate_id 361
rdma_notify 372
rdma_reject 366
converter subroutines
inet_net_ntop 212
inet_net_pton 213
inet_ntop 218
inet_pton 219
Create event channel
open channel 355
CreateIoCompletionPort Subroutine 109
creates a completion event channel 411
current domain names
returning 148
setting 296
current host identifiers
retrieving 157

D

d 111
data
marking outgoing as records 30
Data Link Control 440
Data Link Controls
read subroutine parameters (DLC) 433
data streams
getting position of 9
Data transfer operations
rdma_get_recv_comp 376
rdma_get_request 377
rdma_get_send_comp 378
rdma_post_read 378
rdma_post_readv 379
rdma_post_recv 380
rdma_post_recvv 381
rdma_post_send 381
rdma_post_sendv 382
rdma_post_ud_send 383
rdma_post_write 384
rdma_post_writev 385
data types
receiving GDLC 435
datagram data received routine (DLC) 440
DCE principal mapping 234
default domains
getting 37
searching names 262
Destroying event channel
closes event channel 355
destroys a completion event channel 411
device handlers
decoding name 418
Device management
Device management *(continued)*
attributes of an RDMA port 394
ibv_get_device_list 391
ibv_get_device_name 392
ibv_open_device 393
ibv_query_device 393
ibv_query_gid 396
ibv_query_pkey 395
ibv_query_port 394
Libibverbs library 391
NIC MAC address 396
P_key table 395
rdma_free_devices 387
rdma_get_devices 387
Device Management 387
Device mangement
ibv_get_device_guid 392
discriminated unions
translating 24
DLC
asynchronous event notification 425
extended parameters 431, 435
ioctl operations 426
parameter blocks 433
receiving data
data packet 426
datagram packet 440
network-specific 429
XID packet 439
DLC ioctl operations
DLC_DEL_GRP 441
DLC kernel routines
datagram data received 440
exception condition 425
I-frame data received 426
network data received 429
XID data received 439
DLC subroutine interfaces
close 440
ioctl 428
open 430
readx 434
select 442
writex 437
DLC_DEL_GRP 441
dlcclose entry point 415
dlcconfig entry point 416
dlcioctl entry point 417
dlcmpx entry point 418
dlcopen entry point 419
dlcread entry point 420
dlcselect entry point 422
dlcwrite entry point 423
dn_comp subroutine 111
dn_expand subroutine 112
domain names
compressing 111

E

e 113
eaccept subroutine 113
ebind subroutine 115
econnect subroutine 116

encoding values from 76
endhostent subroutine 118
endhostent_r subroutine 119
ending SNMP communications 84
endnetent subroutine 120
endnetent_r subroutine 121
endnetgrent subroutine 220
endnetgrent_r subroutine 121
endprotoent 122
endprotoent subroutine 122
endprotoent_r subroutine 123
endservent subroutine 124
endservent_r subroutine 125
enrecvfrom subroutine 125
enrecvmsg subroutine 125
entries in the 168, 220
enum values 414
erecv subroutine 125
erecvfrom subroutine 125
erecvmsg subroutine 125
error codes
  using as input to NIS subroutines 44
error strings
  returning pointer 43
esend subroutine 128
esendmsg subroutine 128
esendto subroutine 128
ether_aton subroutine 130
ether_hostton subroutine 130
ether_line subroutine 130
ether_ntoa subroutine 130
ether_ntohost subroutine 130
Event channel operations 355
Event handling operations 373
Event Handling Operations
  RDMA CM event 376
  rdma_ack_cm_event 375
  rdma_event_str 376
  rdma_get_cm_event 373
Event management
  Libibverbs library 411
exception condition routine (DLC) 425
extending base subroutines 81
extending number of entries in 81
eXternal Data Representation 1
external representations, translating from
  arrays 2, 3, 25
  Booleans 2
  C language characters 5, 21
  C language enumerations 7
  C language floats 7
  C language integers 11
  C language long integers 12
  C language numbers 27
  C language short integers 19
  C language strings 20
  C language unsigned integers 21
  C language unsigned long integers 22
  C language unsigned short integers 23
  discriminated unions 24
  opaque data 12

F

f 131
FrcaCacheCreate subroutine 131
FrcaCacheDelete subroutine 133
FrcaCacheLoadFile Subroutine 133
FrcaCacheUnloadFile Subroutine 136
FrcaCtrlCreate Subroutine 137
FrcaCtrlDelete Subroutine 140
FrcaCtrlLog Subroutine 141
FrcaCtrlStart Subroutine 142
FrcaCtrlStop Subroutine 143
freeaddrinfo subroutine 144
freeing 85
from host byte order 191

G

g 144
GDLC
  asynchronous criteria 422
  descriptor readiness 442
  ioctl operations 426
  providing data link control 435
  reading receive application data 434
  reading receive data from 420
  sending application data 437
  transferring commands to 428
  writing transmit data to 423
GDLC channels
  allocating 418
  closing 415
  disabling 440
  opening 419
GDLC device manager
  closing 440
  configuring 416
  issuing commands to 417
  opening 430
GDLC device manager entry points
  dlcclose 415
  dlcconfig 416
  dlcioctl 417
  dlcmpx 418
  dlcopen 419
  dlcread 420
  dlcselect 422
  dlcwrite 423
Generic Data Link Control 440
get_auth_method subroutine
  authentication methods 147
getaddrinfo subroutine 144
getdomainname subroutine 148
gethostbyaddr subroutine 149
gethostbyaddr_r subroutine 150
gethostbyname subroutine 152
gethostbyname_r subroutine 154
gethostent 155
gethostent subroutine 155
gethostent_r 156
gethostent_r subroutine 156
gethostid subroutine 157
gethostname subroutine 157
getipv4sourcefilter Subroutine 316

GetMultipleCompletionStatus Subroutine 158
getnameinfo subroutine 162
getnetbyaddr subroutine 163
getnetbyaddr_r 164
getnetbyaddr_r subroutine 164
getnetbyname 165
getnetbyname subroutine 165
getnetbyname_r subroutine 166
getnetent subroutine 167
getnetent_r 168
getnetent_r subroutine 168
getnetgrent subroutine 220
getnetgrent_r subroutine 168
getpeername subroutine 169
getprotobyname 171
getprotobyname subroutine 171
getprotobyname_r subroutine 172
getprotobynumber 173
getprotobynumber subroutine 173
getprotobynumber_r 174
getprotobynumber_r subroutine 174
getprotoent subroutine 175
getprotoent_r 176
getprotoent_r subroutine 176
GetQueuedCompletionStatus Subroutine 177
gets asynchronous events 412
getservbyname 178
getservbyname subroutine 178
getservbyname_r subroutine 180
getservbyport 181
getservbyport subroutine 181
getservbyport_r 182
getservbyport_r subroutine 182
getservent 183
getservent subroutine 183
getservent_r subroutine 184
getsmuxEntrybyidentity subroutine 72
getsmuxEntrybyname subroutine 72
getsockname subroutine 185
getsockopt subroutine 186
getsourcefilter 316
group network
  entries in the
    handling 303

H

h 191
handling 168, 220
host machines
  setting names 300
  setting unique identifiers 299
htonl 191
htonl subroutine 191
htonll 192
htonll subroutine 192
htons 193
htons subroutine 193

I

i 194
I-frame data received routine for DLC 426
I/O Completion Port (IOCP) Kernel Extension
  CreateCompletionPort 109
  GetMultipleCompletionStatus 158
  GetQueuedCompletionStatus 177
  PostQueuedCompletionStatus 238
  ReadFile 254
  WriteFile 350
ibv_ack_async_event 412
ibv_attach_mcast 404
ibv_create_comp_channel 411
ibv_create_cq 405
ibv_destroy_comp_channel 411
ibv_destroy_cq 405
ibv_detach_mcast 404
ibv_event_type_str 414
ibv_get_async_event 412
ibv_reg_mr 410
if_freenameindex 194
if_freenameindex subroutine 194
if_indextoname subroutine 194
if_nameindex subroutine 195
if_nametoindex subroutine 196
incoming connections
  limiting backlog 235
incoming messages alert 90
inet_addr subroutine 208
inet_Inaof subroutine 210
inet_makeaddr subroutine 211
inet_net_ntop subroutine 212
inet_net_pton subroutine 213
inet_netof subroutine 214
inet_network subroutine 215
inet_ntoa subroutine 217
inet_ntop subroutine 218
inet_ntop6_zone 196
inet_ntop6_zone subroutine 196
inet_pton subroutine 219
inet_pton6_zone 197
inet_pton6_zone subroutine 197
inet6_is_srcaddr Subroutine 198
inet6_opt_append Subroutine 199
inet6_opt_find Subroutine 200
inet6_opt_finish Subroutine 201
inet6_opt_get_val 201
inet6_opt_get_val Subroutine 201
inet6_opt_init 202
inet6_opt_init Subroutine 202
inet6_opt_next Subroutine 203
inet6_opt_set_val Subroutine 203
inet6_rth_add Subroutine 204
inet6_rth_getaddr Subroutine 204
inet6_rth_init Subroutine 205
inet6_rth_reverse Subroutine 206
inet6_rth_segments Subroutine 206
inet6_rth_space Subroutine 207
initializing logging facility variables 73
initiating SMUX peers 86
innetgr subroutine 220
Internet addresses
  converting 208
  converting to ASCII strings 217
  returning network addresses 210
  searching 262

Internet numbers
converting Internet addresses 208
converting network addresses 215
ioctl BPF Control Operations 352
ioctl commands 221
ioctl operations (DLC) 426
ioctl socket control operations 221
ioctl subroutine interface for DLC devices 428
IP addresses
constructing 211
isinet_addr Subroutine 232
ISODE library
logging subroutines 74
isodetailor subroutine 73

K

key-value pairs
returning first 36
keys
searching for associated values 38
kvalid_user subroutine 234

L

Libibverb library 410
Libibverbs library 405
list of 295
listen subroutine 235
ll_dbinit subroutine 74
ll_hdinit subroutine 74
ll_log subroutine 74
local host names
retrieving 157
long byte quantities
retrieving 96
long integers, converting
from host byte order 192
from network byte order 236, 237
to host byte order 236, 237
to network byte order 192
LSs
receiving GDLC 435

M

Management Information Base (MIB)
registering a section 87
manipulating entries 81
manipulating the 79
mapping
Ethernet number 130
master servers
returning machine names 37
memory
freeing 8
memory management subroutines
getaddrinfo 144
getnameinfo 162
memory region 410
Memory region management
rdma_dereg_mr 388
rdma_reg_msgs 388
Memory region management *(continued)*
rdma_reg_read 389
rdma_reg_write 390
message replies 1, 17, 18
MIB list 85
MIB variables 76, 83
multicast addresses
removing 441

N

n 236
name servers
creating packets 263
creating query messages 263
retrieving responses 273
sending queries 273
name2inst subroutine 93
names
binding to sockets 104
network addresses
converting 215
returning 210
returning network numbers 214
network data received routine (DLC) 429
network entries
retrieving 167
retrieving by address 163
retrieving by name 166
network host entries
retrieving by address 149, 150
retrieving by name 152, 154
network host files
opening 297, 298
Network Information Service 33
Network Information Services+ (NIS) 33
Network Information Services+ (NIS+) 45
next2inst subroutine 93
nextot2inst subroutine 93
NIS maps
changing 42
returning order number 41
NIS master servers
returning machine names 37
NIS subroutines
yp_all 33
yp_bind 35
yp_first 36
yp_get_default_domain 37
yp_master 37
yp_match 38
yp_next 39
yp_order 41
yp_unbind 41
yp_update 42
yperr_string 43
ypprot_err 44
nis_add_entry 45
nis_first_entry 48
nis_list 52
nis_local_directory 56
nis_lookup 57
nis_modify_entry 60
nis_next_entry 63

nis_perror 67
nis_sperror 71
nis-remove_entry 67
NIS+ API 45
ntohl subroutine 236
ntohll subroutine 237
ntohs subroutine 238

O

o_ subroutines 76
o_generic subroutine 76
o_igeneric subroutine 76
o_integer subroutine 76
o_ipaddr subroutine 76
o_number subroutine 76
o_specific subroutine 76
o_string subroutine 76
object identifier data structure 79
object tree (OT) 85
ode2oid subroutine 79
OID
    converting text strings to 95
OID (object identifier data structure) 79
oid_cmp subroutine 79
oid_cpy subroutine 79
oid_extend subroutine 81
oid_free subroutine 79
oid_normalize subroutine 81
oid2ode subroutine 79
oid2ode_aux subroutine 79
oid2prim subroutine 79
opaque data
    translating 12
open subroutine interface (DLC) 430
open subroutine, parameters (DLC) 431
opening 183
openx subroutine
    parameters (DLC) 431

P

Packet Capture 352
parameter blocks (DLC) 433
peer entries 72
peer responsibility level 87
peer socket names
    retrieving 169
placing short byte quantities 98
port mappings
    describing 15
portmap procedures
    describing parameters 14
PostQueuedCompletionStatus Subroutine 238
prim2oid 79
processes
    managing socket descriptors 41
Protection domain management
    ibv_alloc_pd 410
    ibv_dealloc_pd 410
protocol data unit (PDU) 84, 89, 90
protocol entries
    retrieving 175
protocol entries *(continued)*
    retrieving by name 172
psap.h file 79

Q

queries
    awaiting response 270
querying 186
Queue pair (QP) management
    rdma_create_qp 386
    rdma_destroy_qp 386
    releases QP 386
Queue pair management
    ibv_create_qp 397
    ibv_destroy_qp 397
    ibv_modify_qp 398
    ibv_post_recv 401
    ibv_post_send 402
    Libibverbs library 397
    work requests 401
Queue Pair management
    ibv_query_qp 403

R

r 240
rcmd subroutine 240
rcmd_af Subroutine 241
rdma_accept 365
rdma_cm 356
rdma_create_id 359
rdma_disconnect 367
rdma_listen 365
ReadFile Subroutine 254
reading 183
reading a MIB variable structure into 82
reading the smux_errno variable 85
readobjects subroutine 82
readx subroutine interface for devices (DLC) 434
records
    input streams
        moving position 31
    marking outgoing data as 30
    skipping 31
recv subroutine 255
recvfrom subroutine 257
recvmsg subroutine 259
registering an MIB tree for 87
remote hosts
    executing commands 240
    starting command execution 274
reporting errors to log files 74
res_init subroutine 262
res_mkquery subroutine 263
res_ninit subroutine 265
res_query subroutine 267
res_search subroutine 270
res_send subroutine 273
retrieving 155, 156, 168, 176
retrieving by address 164
retrieving by name 165, 171, 178, 180
retrieving by number 173, 174

retrieving by port 182
retrieving host entries 155, 156
retrieving network entries 164, 165, 168
retrieving protocol entries 171–176
retrieving service entries 178, 181
retrieving variables 93
Returned error rules
  Libibverbs library 391
rexec subroutine 274
rexec_af Subroutine 276
RPC authentication messages 13
RPC authentication subroutines
  xdr_authunix_parms 27
RPC call header messages 4
RPC call messages 5
RPC message replies 1, 17, 18
RPC reply messages
  encoding 1
RPC subroutines
  receiving XDR subroutines 25
  xdr_accepted_reply 1
  xdr_callhdr 4
  xdr_callmsg 5
  xdr_opaque_auth 13
  xdr_pmap 14
  xdr_pmaplist 15
  xdr_rejected_reply 17
  xdr_replymsg 18
rresvport subroutine 277
rresvport_af Subroutine 278
ruserok subroutine 280

S

s 281
s_generic subroutine 83
SAPs
  receiving GDLC 435
SCTP subroutines
  sctp_opt_info 281
  sctp_peeloff 282
sctp_opt_info subroutine 281
sctp_peeloff subroutine 282
select subroutine interface (DLC) 442
send subroutine 284
send_file
  send the contents of file through a socket 290
send_file subroutine
  socket options 290
sending 89
sending an open 90
sending an open PDU 90
sending traps to SNMP 91
sendmsg subroutine 286
sendto subroutine 288
server query mechanisms
  providing interfaces to 267
service entries
  retrieving by port 181
service file entries
  retrieving 183, 184
set_auth_method subroutine 295
setdomainname subroutine 296
sethostent subroutine 297
sethostent_r subroutine 298
sethostid subroutine 299
sethostname subroutine 300
setipv4sourcefilter 316
setnetent subroutine 301
setnetent_r subroutine 302
setnetgrent subroutine 220
setnetgrent_r subroutine 303
setprotoent subroutine 304
setprotoent_r subroutine 305
setservent subroutine 305
setservent_r subroutine 306
setsockopt subroutine 307
setsourcefilter 316
setting variable values 83
short byte quantities
  retrieving 97
short integers, converting
  from host byte order 193
  from network byte order 238
  to host byte order 238
  to network byte order 193
shutdown subroutine 317
Simple Network Management Protocol (SNMP) 71
SLP subroutines
  SLPAttrCallback 319
  SLPClose 319
  SLPEscape 321
  SLPFindAttrs 321
  SLPFindScopes 322
  SLPFindSrvs 323
  SLPFindSrvTypes 324
  SLPFree 325
  SLPGetProperty 325
  SLPOpen 326
  SLPParseSrvURL 327
  SLPSrvTypeCallback 329
  SLPSrvURLCallback 330
  SLPUnescape 331
SLPAttrCallback subroutine 319
SLPClose subroutine 319
SLPDereg subroutine 320
SLPEscape subroutine 321
SLPFindAttrs subroutine 321
SLPFindScopes subroutine 322
SLPFindSrvs subroutine 323
SLPFindSrvTypes subroutine 324
SLPFree subroutine 325
SLPGetProperty subroutine 325
SLPOpen subroutine 326
SLPParseSrvURL subroutine 327
SLPReg subroutine 328
SLPRegReport callback subroutine 329
SLPSrvTypeCallback subroutine 329
SLPSrvURLCallback subroutine 330
SLPUnescape subroutine 331
SMUX
  communicating with the snmpd daemon 86
  initiating transmission control protocol (TCP) 86
  retreiving peer entries 72
  setting debug level for subroutines 86
smux_close subroutine 84
smux_error subroutine 85
smux_free_tree subroutine 85

smux_init subroutine 86
smux_register subroutine 87
smux_response subroutine 89
smux_simple_open subroutine 90
smux_trap subroutine 91
smux_wait subroutine 92
smux.h file 85
SNMP multiplexing peers 72
snmpd daemon 90
snmpd.peers file 72
socket connections
  accepting 99
  listening 235
socket names
  retrieving 185
socket options
  setting 307
socket receive operations
  disabling 317
socket send operations
  disabling 317
socket subroutine 332
socket subroutines
  freeaddrinfo subroutine 144
  if_indextoname subroutine 194
  if_nameindex subroutine 195
  if_nametoindex subroutine 196
  inet6_opt_append 199
  inet6_opt_find 200
  inet6_opt_finish 201
  inet6_opt_next 203
  inet6_opt_set_val 203
  inet6_rth_add 204
  inet6_rth_getaddr 204
  inet6_rth_init 205
  inet6_rth_reverse 206
  inet6_rth_segments 206
  inet6_rth_space 207
  rcmd_af 241
  rexec_af 276
  rresvport_af 278
socketpair subroutine 334
sockets
  creating 332
  initiating TCP for SMUX peers 86
  managing 349
  retrieving with privileged addresses 277
Sockets 95
sockets kernel service subroutines
  accept 99
  bind 104
  dn_comp 111
  getdomainname 148
  gethostid 157
  gethostname 157
  getpeername 169
  getsockname 185
  getsockopt 186
  listen 235
  recv 255
  recvfrom 257
  recvmsg 259
  send 284
  sendmsg 286
sockets kernel service subroutines *(continued)*
  sendto 288
  setdomainname 296
  sethostid 299
  sethostname 300
  setsockopt 307
  shutdown 317
  socket 332
  socketpair 334
sockets messages
  receiving from connected sockets 255
  receiving from sockets 257, 259
  sending through any socket 286
sockets network library subroutines
  _getlong 96
  _getshort 97
  _putlong 97
  dn_expand 112
  endhostent 118
  endhostent_r 119
  endnetent 120
  endnetent_r 121
  endnetgrent_r 121
  endprotoent_r 123
  endservent 124
  endservent_r 125
  gethostbyaddr 149
  gethostbyaddr_r 150
  gethostbyname 152
  gethostbyname_r 154
  getnetbyaddr 163
  getnetbyname_r 166
  getnetent 167
  getprotobyname_r 172
  getprotoent 175
  getservbyname_r 180
  getservent_r 184
  inet_addr 208
  inet_Inaof 210
  inet_makeaddr 211
  inet_netof 214
  inet_network 215
  inet_ntoa 217
  ntohl 236
  ntohll 237
  ntohs 238
  rcmd 240
  rds 243
  rds-info 247
  rds-ping 250
  rds-rdma 250
  res_init 262
  res_mkquery 263
  res_query 267
  res_search 270
  res_send 273
  rexec 274
  rresvport 277
  ruserok 280
  sethostent 297
  sethostent_r 298
  setnetent 301
  setnetent_r 302
  setprotoent 304

sockets network library subroutines *(continued)*
setprotoent_r 305
setservent 305
setservent_r 306
socks5_getserv Subroutine 335
socks5tcp_accept Subroutine 338
socks5tcp_bind Subroutine 340
socks5tcp_connect Subroutine 343
socks5udp_associate Subroutine 345
socks5udp_sendto Subroutine 347
splice subroutine 349
sprintoid subroutine 79
str2oid subroutine 79
string conversions 95
structures
providing pointer chasing 15, 16
serializing null pointers 15
Supported verbs 355
Supported Verbs
Libibverbs library 391

T

text2inst subroutine 93
text2obj subroutine 95
text2oid subroutine 95
to network byte order 191
traps 91

U

unconnected sockets
receiving messages 257
sending messages 286, 288
unions
translating 24
unique identifiers
retrieving 157
UNIX credentials
generating 27
unregistered trees 85

V

variable bindings 76
variable initialization 73

W

waiting for a message 92
write subroutine, parameters (DLC) 435
WriteFile Subroutine 350
writex subroutine interface (DLC) 437
writex subroutine, parameters (DLC) 435

X

XDR library filter primitives
xdr_array 2
xdr_bool 2
xdr_bytes 3
xdr_char 5
xdr_double 27
XDR library filter primitives *(continued)*
xdr_enum 7
xdr_float 7
xdr_int 11
xdr_long 12
xdr_opaque 12
xdr_reference 16
xdr_short 19
xdr_string 20
xdr_u_char 21
xdr_u_int 21
xdr_u_long 22
xdr_u_short 23
xdr_union 24
xdr_vector 25
xdr_void 25
xdr_wrapstring 26
XDR library non-filter primitives
xdrrec_endofrecord 30
xdrrec_skiprecord 31
xdrstdio_create 32
XDR streams
changing current position 18
containing long sequences of records 29
destroying 6
initializing 32
initializing local memory 28
returning pointer to buffer 10
XDR subroutines
supplying to RPC system 25
xdr_accepted_reply subroutine 1
xdr_array subroutine 2
xdr_authunix_parms subroutine 27
xdr_bool subroutine 2
xdr_bytes subroutine 3
xdr_callhdr subroutine 4
xdr_callmsg subroutine 5
xdr_char subroutine 5
xdr_destroy macro 6
xdr_double subroutine 27
xdr_enum subroutine 7
xdr_float subroutine 7
xdr_free subroutine 8
xdr_getpos macro 9
xdr_hyper subroutine 10
xdr_inline macro 10
xdr_int subroutine 11
xdr_long subroutine 12
xdr_opaque subroutine 12
xdr_opaque_auth subroutine 13
xdr_pmap subroutine 14
xdr_pmaplist subroutine 15
xdr_pointer subroutine 15
xdr_reference subroutine 16
xdr_rejected_reply subroutine 17
xdr_replymsg subroutine 18
xdr_setpos macro 18
xdr_short subroutine 19
xdr_string subroutine 20, 26
xdr_u_char subroutine 21
xdr_u_int subroutine 21
xdr_u_long subroutine 22
xdr_u_short subroutine 23
xdr_union subroutine 24

xdr_vector subroutine 25
xdr_void subroutine 25
xdr_wrapstring subroutine 26
xdrmem_create subroutine 28
xdrrec_create subroutine 29
xdrrec_endofrecord subroutine 30
xdrrec_eof subroutine 31
xdrrec_skiprecord subroutine 31
xdrstdio_create subroutine 32
XID data received routine for DLC 439

Y

yp_all subroutine 33
yp_bind subroutine 35
yp_first subroutine 36
yp_get_default_domain subroutine 37
yp_master subroutine 37
yp_match subroutine 38
yp_next subroutine 39
yp_order subroutine 41
yp_unbind subroutine 41
yp_update subroutine 42
ypbind daemon
    calling 35
yperr_string subroutine 43
ypprot_err subroutine 44